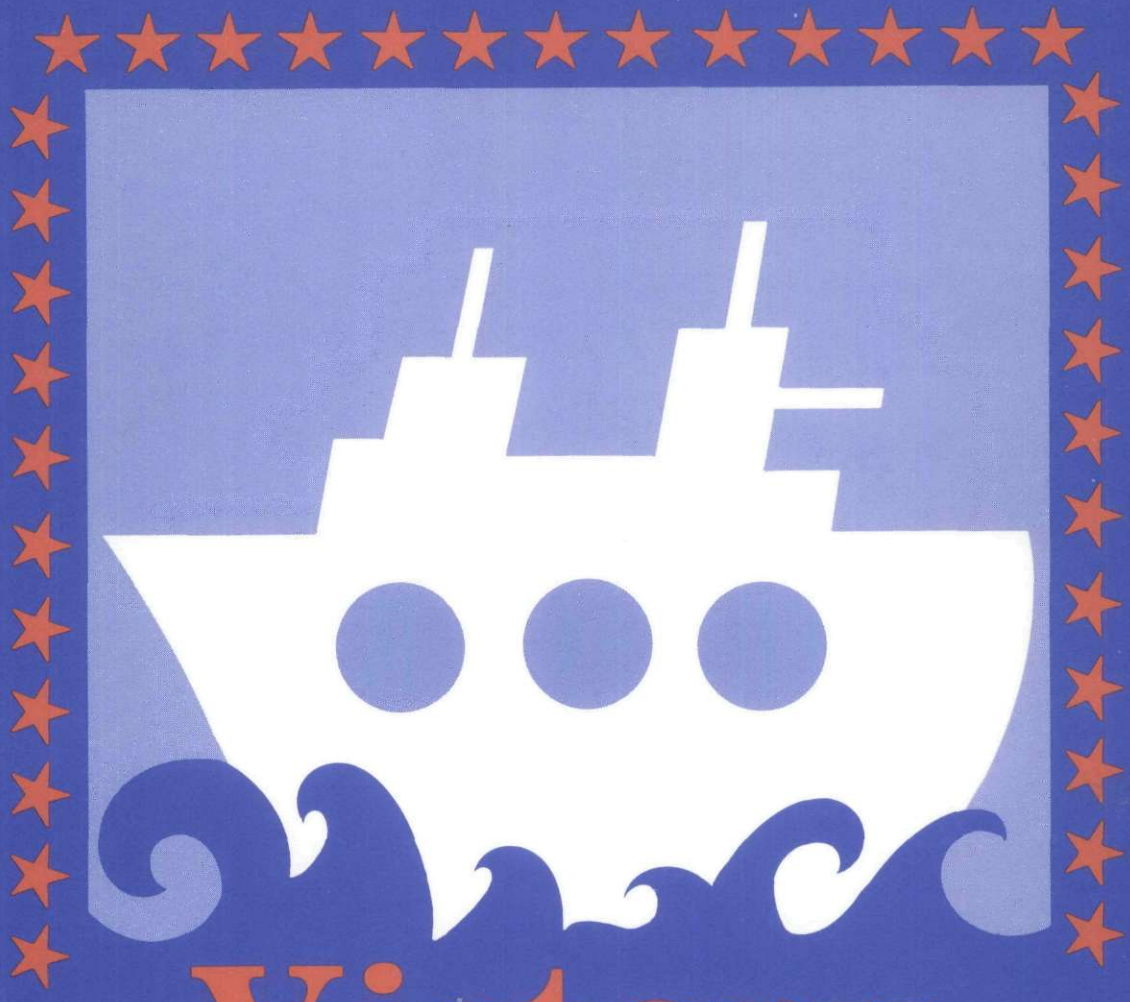
programmer's journal

The Resource Journal for IBM PC Programmers



Victory at C

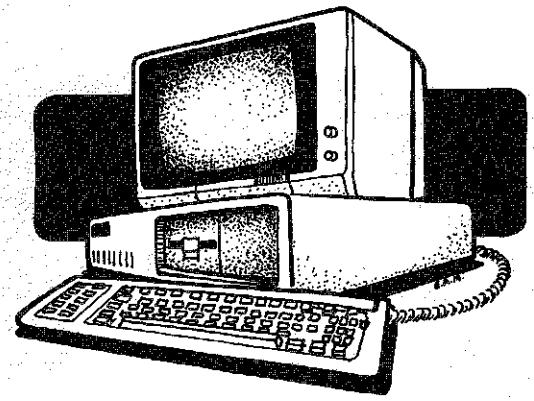
starring
★ Dennis Ritchie ★

and a special interview with
★ Brian Kernighan ★

Volume 2 Number 2

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PROGRAMMER'S JOURNAL



CONTENTS

HOUSEKEEPING 2

LETTERS 3

PJ PROFILE: An interview with Brian Kernighan 4
The next step after C is C++

PJ FORUM: TRYING TO MAKE AN ELEPHANT DO BREAK DANCING by Frank D. Greco 6
What is really happening in the PC Unix look-a-like contest

C: C WHAT'S ON THE STACK by Don Killen 10
Understanding C through Assembly language

TOOLS: OPTIMIZING BASIC COMPILER SOURCE CODE by Wat Buchanon 16
Automatic line number optimizing

ASM: ASSEMBLY LANGUAGE TUTORIAL by Joshua Auerbach 20
What you don't need to know to learn assembly language

C: DATE, TIME AND FREE SPACE by Ted Reuss 27
A series of utility routines in C and Assembly

BASIC: FULL-SCREEN INPUT ROUTINES by Ron Moak 31
Gathering data the easy way

C: MUSIC IN C by Robert T. McCrory 35
Playing Looney Tunes using a C music program

dBASE II: A PRINTER CONTROL DATABASE by Jim Bohle 40
Printer control codes at your finger tips

PASCAL: TOOLS, TOOLS2, AND VIEW MANAGER a review by Chet Floyd 46
Professional routines for application development

C LIBRARY: C PROGRAMMING SUPPORT PRODUCTS 50
A cornucopia of software development tools

MISC 55

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

HOUSEKEEPING

Interest in the C language is definitely picking up. I was told that the yearly Uniforum conference (dedicated to Unix related topics) had 8,000 attendees. Next year it will be in Texas with a projected figure of 35,000. IBM will have a booth there, also.

Besides PJ's usual language and software coverage, we have some articles especially related to C. The interview with Brian Kernighan, co-author of THE C PROGRAMMING LANGUAGE, has brought to light new directions in the world of C.

The list of support software for programmers is increasing monthly. If these packages are used judiciously, they offer tremendous savings in development time. Not only are they fully debugged routines, but the source code is very tutorial in nature. You could spend weeks or months writing assembly procedures to create utility routines merely because the high-level language you use failed to provide them. We shouldn't be forced to acquire hardware and MASM expertise just to access everyday system information.

Looking at someone else's source code is still the best teacher of all. A well written routine can serve many masters. If you can SEE what is going on, you can alter it to serve your own devices.

— Greg Estes

We are compiling a list of the 20 most desirable functions a programmer must create to get the job done. If you have created a function that you could not bear to live without, send it in so that others can share your labor.

PROGRAMMER'S JOURNAL

Publisher: Avant-Garde Creations, Inc.

Business Office:
Avant-Garde Creations, Inc.
1907 Garden Ave.
Eugene, OR 97403
(503) 345-3043

Editorial Office:
2765 Potter St.
Eugene, OR 97405
(503) 687-0398

Subscriptions and Inquiries:
P.O. Box 30160, Eugene, OR 97403
(503) 345-3043

Programmer's Journal. Volume 2, Number 2.

Copyright (C) 1984 by Avant-Garde Creations, Inc.

Published bi-monthly.

Postmaster send address changes to:
Programmer's Journal
P. O. Box 30160
Eugene, OR 97403
(503) 345-3043

Publisher: Mary Carol Smith
Editor: Greg Estes
Managing Editor: Steve Robinson
Associate Managing Editor: Louise Shimmel
Contributing Editor: Jim Bohle
Production: Christine Baldwin
Copy Editing: Marilyn Reaves

Contributing Writers:
Ted Reuss
Ron Moak
Frank Greco
Joshua Auerbach
Chet Floyd
Jim Bohle
Don Killen
Robert McCrory
Wat Buchanon

Subscriptions: Bi-monthly rates are $24 U.S.
($30 in Canada, and $40 elsewhere.)

IBM and Personal Computer are trademarks of
International Business Machines, Armonk, New York
dBASE II is a trademark of Ashton-Tate.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

LETTERS

ARAMCO

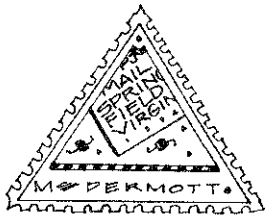
I enjoy your magazine a great deal, and have more faith in your product reviews than those in most of the big glossies. Keep the good stuff coming!!!

James E. Perry
Aramco Dhahran,
Saudi Arabia

BACK ISSUES

I have been purchasing your fine magazine at bookstores and am very pleased with the level of information that it contains. I am unable to subscribe at this time for various reasons, but would like to complete my collection of all back issues. I have numbers 3,4, & 5. I need numbers 1 & 2.

John E. McDermott
Springfield, VA



HIDDEN TREASURE

I discovered PJ #6 hiding between Byte and PC at the local bookstore. What an excellent journal! Both content and style seem to fit my needs and tastes. Just one plea — stay heavy on the programming and light on the reviews. So many of the magazines are becoming glorified "Consumer Software Reports" and (one suspects) marketing vehicles for new packages. You can take your place beside Dr. Dobb's and PC Tech as journals which tell a programmer something really useful and are not readily available elsewhere.

Bernard Argent
Columbia, MD

PATCHER UPDATE

I'm presenting here another version of the patch to the DOS 2.0 COMMAND.COM. The original version had problems with RAM disk software that requires you to set your system configuration switches to indicate more drives than are physically present (such as AST). This version works with AST and other drives. I haven't developed a version of this patch for DOS 1.1, since all real programmers are now using 2.0 or 2.1!

The 2.0 patches also work with DOS 2.1, so no revisions will be required for a while.

Due to a bug in the 2.0 BASIC interpreter, the PATCHER.BAS program may occasionally not work. If you open a file as random and put records out to it, BASIC sometimes gets confused and thinks the length of the file is the same as the offset of the last record put to the file. I've reported this to Microsoft but haven't heard back from them yet. In the meantime, if you encounter this bug, try commenting out line 280 in PATCHER.BAS.

```
A:COMMAND.COM, 18635 ,""Reload from C -- DOS 2.00 for AST""
 93 , 0 , 137                115 , 0 , 7
 94 , 0 , 211                116 , 0 , 67
 95 , 0 , 128                117 , 0 , 198
 96 , 0 , 62                 118 , 0 , 6
 97 , 0 , 127                119 , 0 , 127
 98 , 0 , 1                  120 , 0 , 1
 100 , 0 , 116               122 , 0 , 233
 101 , 0 , 3                 123 , 0 , 7
 102 , 0 , 198               124 , 0 , 5
 103 , 0 , 7                 125 , 0 , 233
 104 , 0 , 65                126 , 0 , 244
 105 , 0 , 198               127 , 0 , 4
 106 , 0 , 6                 169 , 128 , 92
 107 , 0 , 127               393 , 128 , 92
 108 , 0 , 1                 732 , 128 , 92
 109 , 0 , 1                 1392 , 205 , 233
 110 , 0 , 205               1393 , 33 , 234
 111 , 0 , 33                1394 , 115 , 250
 112 , 0 , 114               1395 , 16 , 144
 113 , 0 , 11                2946 , 128 , 92
 114 , 0 , 198
```

Brett Salter
Data Base Decisions

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PJ PROFILE

C AND C PLUS PLUS

an interview with Brian Kernighan, co-author of *The C Programming Language*

PJ: In what area do you do most of your work at Bell Labs?

BRIAN: I'm in the Computing Science Research Center here, and most of my activities deal with C, Unix, and the associated document preparation. Any conventional programming that I need to do is written in C.

PJ: Are you involved in any new books?

BRIAN: I just finished a book with Rob Pike, called *The Unix Programming Environment*. Believe it or not, in between writing books I actually have to do some regular work.

PJ: Since your 1978 book, *The C Programming Language*, with Dennis Ritchie, have there been any attempts to redesign C?

BRIAN: I think the proper way to look at it is that C is undergoing a steady, though not very fast, evolution. There is an ANSI Standards committee working on the standardization of C. Bell Labs have representatives on the committee, as do other groups interested in C. I conjecture that, even there, the changes that get made in the language that you and I know today will be fairly modest.

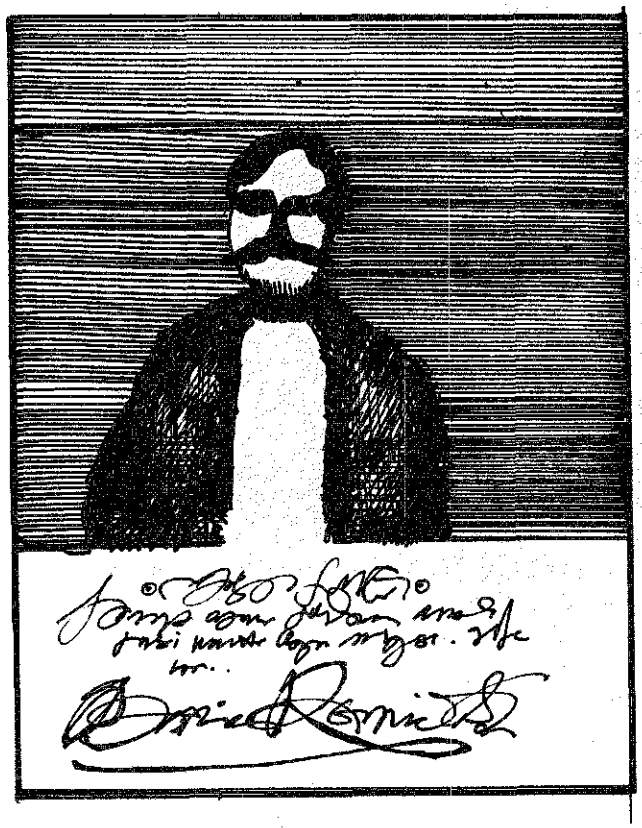
Over the years, changes to C have been directed toward making the language more type safe, making it easier to say precisely what you mean, and making it more portable. Portability is one of its strengths. Also, making it easier for the compiler to check what you are doing is sensible. As an example, there is a fairly strong movement afoot to put in some kind of declaration of the types of function arguments at the point where the function is used, as well as at the point where the function is defined. Enhanced cross-checking is definitely desirable.

There has, over the years, been a tightening-up of what you can do with pointers without saying, "I'm about to break the rules." These kinds of clarifications will no doubt continue, but there will probably be no fundamental change in C, and certainly nothing that would look like a revolution.

PJ: Are there any "enhanced" versions of C around?

BRIAN: There is, inside in the Labs, a fair amount of work going on concerning a language that might be considered that next step up from C. A language that is called C Plus Plus (C++). Basically, it is C with mechanisms for data abstraction, and a fair amount of notational convenience to go along with it. Bits and pieces of C++ have been described to the outside world, and I suspect there will be more coming out in the next few months.

PJ: Would you call it an extension of C or a mild redesign?



BRIAN: It is definitely a compatible extension. You can still program in C and it will look like it did before, but you can take advantage of the new facilities, like data abstraction. Mechanisms of this kind are analogous to the package mechanism in Ada or the module in Modula, but in fact they are just the class mechanism of Simula; technically, that's what it is. So we used to call the extension "C with glasses." That name didn't grab me, so it became C++.

PJ: I think the first burning question about C++ is, "Will my old C programs compile OK?"

BRIAN: With a couple of trivial exceptions, yes. There are some new keywords. If you have used any of them, then they must be changed. One of the key words is "class."

You do not have to learn a new way of thinking with C++ or learn a new set of syntax. The entire body of C code that exists already will remain working. Programmers can extend their focus to data abstraction, and back into it gently.

PJ: What other keywords are included?

BRIAN: There is an "inline" keyword that will allow you to define inline functions, the functions will be inserted inline. This is more powerful than a macro processor approach. Another keyword is "generic," useful for defining functions which allow the compiler to pick the appropriate function body to complete the task. An example would be a generic operator "plus" that would handle floating addition or matrix addition with the same notation. The compiler figures it out. Generic is used as it would be in Ada.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PJ: How is C++ implemented?

BRIAN: It is just a C program. It can run on any machine with a C compiler. Today's implementation — that is literally the one we are using this minute — it produces vanilla C as its output. You are, in effect, doing two translations of the program at the source level, not a good idea in a production environment. At some point it will create an intermediate code. It is definitely a full-blown compiler and not just a pre-processor. You get error messages that are, thank goodness, appropriate for the program its working on.

PJ: Is additional information available about C++?

BRIAN: I don't think that anything has been published yet. There are a couple of papers that are in Bell Labs' internal release mill, and they will probably be out in a couple of months. A preliminary version of the language was described in a paper published in *Software Practice and Experience*.

PJ: I am not familiar with that journal.

BRIAN: For shame, how can you be in the software business and be unaware of the journal? It is a publication from England that is distributed by Wiley of New York.

PJ: Do you think that Unix is feasible on the IBM PC?

BRIAN: It already exists, but I think it really depends on what you are planning to use the computer for. If you intend on solving numerical problems, then it is not going to be right. The environment would not be suited for processing thousands of matrices. If I were going to bw writing programs of medium size, or preparing documents and editing, then it would be qutie functional. The speed problem is not usually a function of the processor chip but of the disk access time. The hardware bottleneck in terms of getting things done is the disk drive. Program development and document preparation are ideal for a Unix environment.

PJ: How has the interest in C and Unix evolved?

BRIAN: Ten years ago Unix was on just a few systems scattered around Bell Labs. They were all run by rather strange people, myself included. It would have taken a fair amount of perceptiveness to imagine that in a few years we could go into Radio Shack and purchase a version. Radio Shack did not even exist at the time. The original Unix system was very successful, partially because the computer it ran on, PDP-11, was an excellent machine. A lot of bang for the buck. A group of people could afford the machine. This was in 1969-1970. Now it is possible for single individuals to purchase a machine that can run Unix. I wouldn't have predicted this ten years ago.

PJ: Do you think the current popularity of Unix is attributable to the operating system or the C language?

BRIAN: I think that originally it was C. It developed a following on the 8-bit machines, 6502, Z-80s and so on. Then Unix became popular. C lets you get at the hardware while still retaining structure and portablility. But Unix was too big to run on an 8-bit computer. The 16-bit machines, 8088 and 68000, have a greater Unix potential.

PJ: Perhaps the assembly language programmers who discover C as a transportable tool are reluctant to let it go?

BRIAN: I have not written any assembly language in over ten years.

PJ: What debugging tools are available for C?

BRIAN: There are a couple of debuggers under Unix. Though not particularly good, they do tend to think at a source code level, in terms of C, rather than in a core dump style mode. Some really innovative work is being done here on debuggers, using bit-mapped terminals and mice.

PJ: When you worked on the C book with Dennis Ritchie, what was the hardest part of the language for you?

BRIAN: Pointers. I really think it is hard to get pointer across. That section of the book could have been done in a different way. I do get calls from people who say, I don't understand pointers. I tell them it is all there in the book, but for some reason it does not communicate the concept. Declarations could be improved. Complicated declarations are a problem. The concept of declaring things like they are going to be used is nice, but it seems to break down for complex ones.

PJ: How deep do you go with pointer declarations?

BRIAN: Never more than two. And probably not many of those. With the exception of "argv," I do not ever use more than one.

PJ: What version of Unix are you running under?

BRIAN: Right here where I sit, and you must remember we are a small and off-in-the-corner part of Bell Labs, we are running the eighth addition. The bulk of Bell Labs is running System V.

PJ: Are newer versions of Unix redesigns or extensions?

BRIAN: Features hardly ever go away if that is what you mean. The biggest feature in the eighth addition is paging, it went through Berkeley and it uses their paging features. Our networking is very different from theirs, as is our terminal handling.
□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

TRYING TO MAKE AN ELEPHANT DO BREAK DANCING

by Frank D. Greco

On January 12, 1984, IBM announced a "new operating environment derived from the Unix* Time Sharing System" for the IBM Personal Computer. It was called the IBM Personal Computer Interactive Executive or PC/IX. At Uniforum, the Unix users group exhibition/symposium in Washington,D.C., IBM's booth was what you would expect: it was large, it drew enormous crowds, it had PCs on display (would you believe on pedestals?), and it was staffed by many three-piece sales types who never did explicitly answer any technical questions. IBM also handed out a PC/IX General Information Manual that was a fairly succinct synopsis of the commands available under PC/IX.

This port of Unix was written by Interactive Systems Corporation of Santa Monica, California. As a matter of fact, PC/IX is based on Interactive's operating system IS/3, which itself is a Unix System III derivative (i.e., a licensed AT&T Technologies product). The cost of PC/IX is $900. During the spring of 1984, AT&T will offer a single-user binary license for sixty dollars and also volume discounts up to 60 percent. In light of AT&T's greatly reduced price for binary licenses for Unix (the source license is still about $43,000), I really do not understand the high price that IBM is asking. My GUESS is that, although IBM cannot turn their heads from the increasing wave of interest in Unix (there is a strong rumor that IBM does its homework when it comes to marketing), they can probably prevent or delay Bell Unix from becoming TOO popular. Let's face it, if Unix got to the point where it truly was a standard, IBM might be in a strategically poor position; AT&T would be controlling the operating system. There is an old maxim in the industry that says "He who controls the operating system controls ALL." I'm sure IBM is a firm believer in that old adage.

Look at what's happening with Microsoft's formerly symbiotic relationship with IBM. IBM has a long history of NOT staying with any vendor for an extended period of time. Microsoft supplies MS-DOS for both the PC and the PCjr, as well as for several other popular machines. That's about all the power that IBM will allow Gates and company to have. IBM wants to get into the software business in a big way, and does not want to help build competitors. I sincerely doubt that IBM will endorse Xenix (which is a REAL Unix) from Microsoft, unless IBM literally buys into Microsoft (like they did with Intel). This is probably why Microsoft has been saying that they expect 50 percent of their 1984 applications software revenue to come from the Apple Macintosh, which is probably not a bad idea financially. Macintosh looks like a hit (even with a current memory maximum of 128K). Microsoft knows they cannot be pilot fish for the great Blue Shark forever (was that a metaphor or was that a metaphor?).

Technically speaking, PC/IX is a SINGLE-USER Multi-Tasking Operating System. Most of the look-alikes are MULTI-USER Multi-Tasking OS's. Why did IBM release a single-user system? Well, if a multi-user Unix was released, nontechnical types might rush right out and buy the thing, thinking that it could handle 2, 3, or 4 concurrent users. We all know that the little 8088 inside the PC, coupled to the relatively slow Seagate Winchester, cannot possibly handle the power that a full (multi-user/multi-tasking) Unix could muster. These novice users might be a little disappointed to see a word-processing package running as slow as a three-legged tortoise taking a leisurely stroll. Or a C compile taking more time than the gestation period of an elephant.

Even worse, the 8088 has no real memory management! This presents real problems for a multi-user Unix system. You might have to reboot every time the assembler freak does his debugging or the C language programmer gets a core dump from a stray pointer. IBM, of course, would not like the negative publicity, so they opted for a single user version. Still, a single user who runs three or four simultaneous compiles should expect some speed degradation.

As any programmer knows, the more users on a system, the slower it runs. The 8088 ain't no 32032! Putting a full Bell Unix on the IBM PC is like restricting a Lamborghini sports car to a 55 mph speed limit. Plenty of comfort and power, but no chance to show it off. Unix aficionados shouldn't despair, from what I've read, some of the look-alikes, such as QNX and Coherent, are reasonably quick and might prove to be viable alternatives to a REAL Unix. Unix was written as machine independent as possible to increase portability among different processors with different machine architectures. This

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

implies that Unix is not as fast as an operating system written specifically for the PC. We will have to wait until PC/IX benchmarks are published to know the truth.

According to IBM the requirements for running PC/IX are:

* 10 Megabyte Hard Disk

* At least one double-sided, double-density floppy drive

* At least 256K RAM, (an IBM representative at the Uniforum show in Washington, D. C., whispered "512K" to me)

* IBM Color monitor or IBM Monochrome monitor (Note that the IBM Color monitor is supported in TEXT mode ONLY!)

* The Asynchronous Communications Support Adapter

PC/IX comes on 19 floppies, to be subsequently installed on the hard disk. Most of the Unix programs and utilities are included. Programmers' tools, such as a C compiler and an 8086/8088 assembler, are of course available. (I wonder if the C compiler follows the original Unix philosophy and generates assembler mnemonics for subsequent processing by the local assembler???) The Bourne Shell is included as the standard and ultimately programmable command interpreter. Hierarchical file systems (mountable), Shell redirection, true pipes (not DOS 2.0 kludges), and process primitives are provided also. A complete collection of archaic tools are there as well: adb (cryptic debugger), ed (outdated line editor), diff and comp (supposed to tell you what is different about two files, try them and then tell me what you think), chmod (the so-called user-friendly style is confusing), and ptx (generates permutated indices - I personally like the old-fashioned types). The powerful but tedious text processors nroff and troff (these are NOT word processors as some misinformed people are saying) are also included. IBM, however, states that the IBM PC printers are not capable of printing documents formatted with some of the 'advanced' features of nroff/troff. Why aren't they? If IBM was as serious about PC/IX as most industry analysts would lead you to believe, don't you think they would improve on Unix's effectiveness? PC/IX, I gather, does not come with device drivers for disk emulation in RAM. That's odd. A few other Unix look-alikes do. Since Unix is heavily disk-bound, putting the most often called programs on a RAM disk would speed things up tremendously. For all you Unix purists, the arcane is there too: awk, yacc, lex, sno, m4, and bs. An excellent screen editor, INed (written by Interactive), comes with the package.

If you want a real Unix on your PC, investigate PC/IX thoroughly before you purchase it. If you want a Unix look-alike, compare QNX, Venix, Coherent, uNETix, and maybe even MS-DOS 3.0 (coming soon, I've heard, and with real pipes and a consistent file structure). They are all multi-user, though I'm not sure about MS-DOS 3.0. QNX and uNETix even have MS-DOS emulators (these do not always work because many programs call the ROM BIOS routines directly and bypass MS-DOS) so you can run your favorite PC programs. I do not think IBM is ready to concede any battle with AT&T. IBM loves to set precedents, not follow them. Think about this: Unix is a natural for networking, and AT&T wants to have ultimate control of the telecommunications network, just as they have control of the telephone system. Do you think IBM wants AT&T to be in the driver's seat? With this PC/IX announcement, IBM seems to be telling the marketplace that they are very interested, but are not completely committed to Unix. But whatever IBM ultimately does in the operating system area, will most likely be in a Unix-like vein. In typical IBM fashion, they might announce some sort of Unix Superset that would lock users into using IBM software and hardware. Those old mainframers out there are probably thinking to themselves, "Where have I heard THAT before!" This new operating system would probably be a kind of MS-DOS/Unix synergistic blend. Of course, a networking version wouldn't hurt either. IBM might even do the sacrilegious and offer a superset of the C language! We know by now that the elephant can indeed tap dance, but the question remains: can he star in the next Flashdance movie?

PCjr: THE GOOD, THE BAD, AND THE UGLY

Recently, the IBM PCjr has caused quite a controversy in the microcomputer industry with its price, compatibility, support, and capability. The Junior has some good features, some bad features, and one extremely ugly feature. First, let's figure out where the market niche for the little beast is. The base price for the Teledyne-manufactured (Lewisburg, Tennessee) IBM PCjr, as quoted in the IBM Product Center Suggested Customer Configuration Order Form (what a mouthful!), is $669. Original PC owners know of IBM's unbundling tactics and realize that it will take a lot more money to make the little guy fully functional. Realistically, if you buy all the parts from IBM, you are going to dish out a minimum of $1,459. This includes a connector for the family television, the IBM PCjr BASIC Cartridge, PC-DOS 2.10, 1-360K disk drive, an 80 column board, and an additional 64K of RAM. The IBM PC Compact printer (thermal) will cost you another $175. You can use your favorite serial printer with the built-in serial interface, but be forewarned, if your printer requires the standard 25 pin connector (whose device doesn't?), you will have to buy the IBM PCjr Adapter Cable for Serial Devices. A standard (Centronics) parallel interface is optional and will wring even more money from your wallet. Including the small 50 cps thermal printer, the PCjr costs a total of $1634.

Considering the price, how can we suggest that the PCjr is in the same market niche as the low-priced micros (VIC-20, Timex/Sinclair, Commodore 64)? If you want to learn about computers, $1634 is a lot of money to spend. The purchase

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

might even turn out to be a quick course in how computer companies make money. With a PC already at the office, you might want to do some work at home using a compatible machine. In this case, the price doesn't seem so bad, especially since it may present a very deductible image.

Even so, I really don't think IBM can sustain such a high price for a machine that is basically designed for the home market when we are starting to see $500 Apple IIe's (dealers must make room for those sexy Macintoshes) and $500 68000-based systems (thanks to good old Uncle Clive). The mass consumer is ALWAYS price conscious, and IBM is always consumer conscious. But for us software developers, it may well prove to be advantageous to buy the PCjr, because most industry experts feel that, barring any distribution shenanigans from IBM the Junior will find many, many homes. This covers the GOOD part.

The BAD has to do with the capabilities of the machine. The PCjr still has the same 8088 processor, thanks to a programming goof that IBM made when it designed and wrote the original BIOS routines. According to many sources, IBM originally wanted to use the Intel 80188 microprocessor in the PCjr. Because the 80188 incorporates some ancillary support chips right on the microprocessor itself, it would eliminate some engineering headaches, reduce the overall cost, and speed up the overall performance of the machine. That makes the 8088 a reasonable and pragmatic choice.

OPTIMIZING C86™ controls Charlie...

Like a Puppet on a String!

Serious programmers can use Optimizing C86 to control the IBM PC and other MS-DOS/PC-DOS computers. Features include:

- A **Full** and **Standard** implementation of the C language balancing tight control of the machine with the productivity and portability of a high level and standard language.
- **Hardware and operating system interfaces** for: graphics, interrupt control, **8087** use, I/O ports, real time applications and for producing ROMS.
- Use of the standard MS-DOS Linker and a compiler option to produce MASM Assembler output. Helps integrate with MASM and other standard languages like MS FORTRAN and PASCAL.
- **A rich set of libraries** that includes source for K&R functions, extras for string handling, graphics, sorting, floating point (8087 and 8086/88), **"Large" model** memory use (1,000K RAM), **"Small" memory** model, MS-DOS 1.1, 1.25, 2.0, 2.++.
- Support for numerous **add-on libraries** including: HALO **Graphics,** C Tools for fundamentals in many areas, PHACT for **ISAM** file management and numerous others. Ask for a list.

Pull Charlie's strings with our fast, complete, proven, reliable C Compiler—the leading compiler for serious programmers of MS-DOS and CPM-86 systems.

Computer Innovations
980 Shrewsbury Avenue
Suite J-950
Tinton Falls, N.J. 07724
(201) 542-5920

"They Say It All... We Do It All!"

Visa and MasterCard accepted.

C86 is a trademark of Computer Innovations, Inc. CPM-86 and MPM-86 are trademarks of Digital Research. MS-DOS is a trademark of Microsoft. PC-DOS is a trademark of International Business Machines.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

But the authors of the BIOS (IBM) used some interrupt vectors that Intel had reserved for future enhancement. Although these reserved interrupts were thoroughly documented by Intel, IBM apparently goofed and used them anyway. Instead of issuing a statement to all software developers to modify ALL calls to these BIOS interrupts (a ludicrous thought, to say the least), IBM sent the chips back to Intel and asked for 8088's instead of the 80188.

Needless to say, Intel had to alter the mask for the 80188 so it would remain object compatible with existing PC software. I'm sure Intel was not exactly happy about the situation, but when IBM is the customer (and part owner) Intel really had no other choice. Microsoft chairman Bill Gates reportedly was fuming about rampant rumors that Microsoft was at fault. In fact, if every developer faithfully used DOS calls like they are supposed to, we would probably have the 80188 PCjr right now. Microsoft would have changed their DOS software and no one would have noticed a thing. But realistically, everyone uses the BIOS routines, which makes the software update solution a moot point. IBM will never admit to this Big Blue Blunder, but when both Microsoft and Intel point the finger at IBM, something is up. What this all means is that we still have the old 8088 and not the speedier 80188. Expecting innovation from IBM is like expecting support from Commodore.

The PCjr comes standard with 40-column display capabilities; the 80-column board plus an additional 64K is optional (i.e., more money). From recent reports, both read and heard, the PCjr is slower than its ubiquitous parent. At a local IBM PC user group meeting, an IBM representative said that the video buffer can start on any memory location and enable page flipping (a la Apple II), resulting in very smooth animation. He hinted that this feature unfortunately might slow some nongraphic programs that do heavy screen output. Apparently, the system must check where the current video buffer resides before performing screen I/O.

Something else that happens to be GOOD about the PCjr is the sound chip from Texas Instruments. This chip allows three concurrent voices and one white-noise channel. The white-noise capability allows a cacophony of buzzes, whirs, and clicks to emanate from the Junior. Game programmers will love it. Perhaps we might see some inexpensive voice synthesizers soon that will use the TI chip. It is a little curious that IBM chose to vastly improve the sound output of the PC but not its graphics capabilities. There is more color present in the PCjr graphics modes than in the older PC, but still only two levels of luminosity. If only IBM had included a graphics chip similar to Atari's GTIA...sigh... all kinds of neat graphic tricks (e.g., depth illusions) would be a snap to create, even with several levels of luminosity. But let's face it, although the PCjr can play some games, it was not designed to be a game machine. If games are at the top of your list, then get a Commodore 64, an Atari, or a Sinclair 2068. They're considerably cheaper, and numerous games are already available for them.

Now the UGLY part. It is the PCjr keyboard. Many criticisms surround the cordless (infrared) controlled keyboard that has a less-than-standard layout. I guess IBM did not want to ruin their image as expert PC keyboard designers. I always thought that IBM was the designer of the Selectric and its standard keyboard layout; maybe that was a rumor. The keys of the PCjr are similar to the keys on cheap portable typewriters. Although the keys are very small, it is not a chicklet keyboard. I don't remember chicklets being that large! (When you start arguing about what a chicklet keyboard IS and what it is NOT, you are either a bona fide microcomputer fanatic or a gum salesman!) I tried a PCjr at a local Computerland the other day and found the keyboard to be surprisingly light in weight. As a touch typist, I found the PCjr keyboard slowed me down a bit, especially when I used the function keys. The keys are not labeled on the keytops; rather, they are labeled on the keyboard itself. So entering simultaneous keys can be annoying. Keytronics will come to the rescue undoubtedly. Two things that I DO like about the keyboard are the large return key (it is hard for your fingers to miss it) the fact that there is no backslash next to the left shift key (Hooray!). Actually, I think that the cordless aspect of the keyboard was added purely for novelty, as most PCjr owners will be working immediately in front of the screen and won't be firing (pseudo) scan codes from across the room. As a bell-and-whistle, I would rate the cordless keyboard as just a bell. But it was fun bouncing the infrared beam first off the cover of a book and then off a person's white shirt to reach the system unit. However, I really don't foresee using a PCjr to study Snell's law.

The IBM PCjr will probably be purchased by existing owners of PCs who want to bring a familiar face home from the office. Imagine an average consumer (not a software developer) using a PC at work and buying an Apple or a Commodore 64 for the kids at home. I personally find the thought of a PC/XT at work and an Apple Macintosh at home very alluring. But, for the uninitiated (the vast majority of PC owners), perhaps two different machines would be a bit confusing ("Mommy, why doesn't 'SCREEN 2' work on the Commodore the way you said it would?"). We can hope that IBM does not pull any crazy distribution tricks, such as purposely holding back quantity release of the PCjr to increase interest and thus reducing the chance of retailers to discount the little guy (making licensed IBM dealers happier with larger profit margins). Assuming there are no distribution problems, we should be seeing a lot more of these little Blue machines. This undoubtedly puts bigger smiles on the faces of software developers, and book authors too.

**Unix is a trademark of AT&T Bell Laboratories*

□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

C WHAT'S ON THE STACK

by Don Killen

Cast, pointer, stack, printf, Heap, alloc — arrgh ! Who are these Kernighan and Ritchie folks, and why do they hate me so much? Perhaps you're a contented ASM programmer who is intrigued by the new kid on the block - C. But after a closer look, you're not convinced you want to explore the language further. You're anxious about leaving the clarity of MASM, and you would even settle for BASICA if your understanding of C doesn't improve. What is a Heap and how is it different from the more familiar stack? What exactly IS a pointer in C and how do I use it, hopefully not at the expense of my sanity ? Why does one compiler alter character values greater than 7F hex while other compilers do not?

If the stack seems a place to push and pop only, stay tuned. We will journey into the bowels of a C program and explore what is on the stack, what some code looks like to the CPU, and what happens when we call a C function. We're going on a Heap hunt.

THE STACK

The stack (in C) is used as a dynamic storage area:

1. Arguments are placed there when one function calls another.
2. Temporary (dynamic) variables are allocated space there.
3. Return address and critical register values are stored there.
4. Temporary place to push and pop.

The stack is addressed by the SS and SP registers; in addition, the BP register is universally used as a frame address pointer (i.e., SS:BP). To illustrate this, let's examine a slice of code as disassembled by the Lattice device OMD.EXE. All examples were coded in C, compiled with Lattice v1.4, then processed by OMD and hand annotated. Each example is first shown in C, followed by the disassembled version in assembly.

Example 1 - Temporary Variables

```
fx()
{
int a;
a = 0x0F0F;
}
```

```
EXTERNAL DEFINITIONS
fx  0000-P
PROGRAM SECTION: 10 BYTES
0000 55                 PUSH    BP
0001 83EC02             SUB     SP,02
0004 8BEC               MOV     BP,SP
0006 C746000F0F         MOV     W,[BP+00],0F0F
000B 83C402             ADD     SP,02
000E 5D                 POP     BP
000F C3                 RET
NO DATA SECTION DEFINED
```

The code says: arrange for a temporary (dynamic) integer variable somewhere (on the stack) and set it to 0F0F hex. If we had manually coded the assembler, we might have said:

```
        public  fx
fx      proc    near
        push    bp              ; save former frame pointer for C
        sub     sp,2            ; allocate 2 bytes on the stack
        mov     bp,sp           ;  and set our frame pointer
        mov     word [ss:bp],0F0Fh  ; set the value
        add     sp,2            ; reset SP for exit
        pop     bp              ; restore register for C
        ret                     ; DONE !
fx      endp
```

This technique of memory allocation on the stack illustrates an important point: once the function exits, the storage locations are released, gone, to be overwritten by the next function to be called. Note that in the above example no mention or use is made of either Heap or static storage. A clue is that neither DS nor ES segment registers were used. (Are we in a Heap of trouble here?)

```
int a;
fx()
{
a = 0x0F0F;
}


data    segment byte            ; note - segment is common
        public  a               ;        but "a" is public


a       dw      0               ; usually compilers zero this
data    ends
```

Example 2 - Static Data

This function is like the preceding one except that no dynamic storage is allocated; instead, the variable "a" exists in a static data segment (see fig. 1). In this case the scope of the variable is not limited to the function.

```
code    segment byte
        public  fx
fx      proc    near            ; -- we're in the small model..
        push    bp              ; just like previous example
        mov     bp,sp
        mov     word [DS:D.0000],0F0Fh  ; offset 0 in data area D
        pop     bp
        ret
fx      endp
```

Here we see a simple "write to an address" at offset 0 in the commonly held data area. The data area is available to any other function compiled within the same file. The compiled C function needlessly goes through the motions of fooling around with SP and BP. It makes arrangements for them even if they are never used.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Example 3 - Static Data, Limited Scope

```
fx()
{
static int a;
a = 0x0F0F;
}
```

You might think — aha, at last we'll see the Heap — but no, the Lattice and CI-86 compilers allocate static storage the same as external integer declarations. The usefulness of a static variable, however, is still confined within the function. It seems to be easier to group these data segments together than to use the ubiquitous Heap.

Example 4 & 5 - External Data

```
extern int a;
fx()
{
a = 0x0F0F;
}


fx()
{
extern int a;
a = 0x0F0F;
}
```

```
fx      proc    near
        push    bp
        mov     bp,sp
        mov     word [DS:a],0F0Fh       ; a is external
        pop     bp
        ret
fx      endp
```

The above two slices produce exactly the same code when compiled with Lattice and CI-86. An extern, it seems, is an extern. This differs from examples 1, 2, and 3, however, in that no knowledge is evident of just where the variable "a" resides, except that it must be in the data segment addressed by the DS register. And the variable is accessible to all functions in the program.

It turns out that there is a collection of data items or short segments in the static area addressed by DS or ES. So there is still no Heap. Statics, register variables, and externs are collected in a Heap-like place in the data area. Every one of a program's numerous functions that declares external (or static) data items forces one or more bytes of memory to be reserved in a pooled data segment. The DOS linker makes the actual memory location assignments based upon the list of names it receives from the source code via the compiler.

Ad Changed 8/5/84

The PROGRAMMER'S SHOP™ helps:

Compare, evaluate, find products. Get Answers. GUARANTEES.

"C" LANGUAGE

		LIST PRICE	OUR PRICE
APPLE: AZTEC C-Full, ASM		$199	call
8080: BDS C-Fast, popular		150	125
8080: AZTEC C-Full		199	call
Z80: ECOSOFT-Fast, Full		250	225
8086: **C86**-optimizer, Meg		395	call
8086: Lattice - New 1.1 & 2.0		600	495
Microsoft (Lattice)	MSDOS	500	call
Digital Research - Megabyte	8086	350	269
Desmet by CWare-Fast	8086	109	99

BASIC

		ENVIRONMENT
Active Trace-debug	8080/86	$ 80 72
MBASIC-80 - MicroSoft	8080	375 255
BASCOM-86 - MicroSoft	8086	395 279
CB-86 - DRI	CPM86	600 439
Prof. BASIC	PCDOS	345 325
BASIC Dev't System	PCDOS	79 72

EDITORS Programming

BELLESOFT - PASCAL	MSDOS	$245	call
C Screen with source	8080/86	NA	60
EDIX	PCDOS	195	149
FINAL WORD	8080/86	300	215
MINCE	CPM, PCDOS	175	149
PMATE - powerful	CPM	195	175
	8086	225	195
VEDIT - full, liked	CPM, PCDOS	150	119
	8086	200	159

SERVICES

* **Programmer's Referral List** * **Dealer's Inquire**
* **Compare Products** * **Newsletter**
* **Help find a Publisher** * **Rush Order**
* **Evaluation Literature free** * **Over 300 products**
* **BULLETIN BOARD** - 7 PM to 7 AM 617-461-0174

PASCAL

	ENVIRONMENT	LIST PRICE	OUR PRICE
PASCAL MT + 86	CPM86/IBM	$400	$289
without SPP	CPM80	350	249
MS PASCAL 86	MSDOS	350	255
SBB PASCAL - great, fast	PCDOS	350	315
PASCAL 64 - nearly full	COM 64	99	89

LANGUAGE LIBRARIES

C to dBASE interface	8080/86	$125	$115
C Tools 1 - String, Screen	PCDOS	NA	115
C Tools 2 - OS Interface	PCDOS	NA	92
C Tools - XOR - Graphics, Screen		NA	95
FLOAT87 - Lattice, PL1	PCDOS	NA	115
GRAPH GSX-80	CPM80	NA	75
HALO	PCDOS	150	125
ISAM Access Manager-86	8086	400	300
PHACT - with C	PCDOS	NA	250
FABS	CPM80	150	135
PASCAL TOOLS	PCDOS	NA	115
SCREEN Display Mgr. 86	8086	500	375
PANEL-86 - many	PCDOS	350	315
WINDOWS - AMBER	PCDOS	295	call

FORTRAN

MS FORTRAN-86 - Meg	MSDOS	$350	$255
SS FORTRAN-86	CPM-86	425	345
FORTRAN-80 - '66 decent	CPM-80	500	350
INTEL FORTRAN-86	IBM PC	NA	1400
DR FORTRAN COMING			
RM FORTRAN COMING			

OTHER PRODUCTS

Microsoft ASM-86	MSDOS	100	85
Microshell improve CPM	8080	150	125
IQ LISP - full, 1000K RAM	PCDOS	175	call
PLINK-86 - overlays		350	315
PL/1-86	8086	750	560
Prolog - with samples	PCDOS	NA	125
MBP Cobol-86 fast	8086	750	695
COBOL - Level II	8086	1600	1275
TRACE-86 - debug	PCDOS	125	115
CODESMITH-86 - debug	PCDOS	195	139

NEW FEATURES For MICROSOFT C

Halo-Graphics	$125
Panel - Screen	$315
PHACT - ISAM	$250
C Tools 1 - String	$115
C Tools 2 - OS Interface	$ 92
Greenleaf - 200 functions	$165
Others coming	

Free Report: PRODUCTIVITY USING MSDOS

Assume use of compiler and typical editor. What commercial or public domain products, what techniques improve productivity? "Productivity with MSDOS" is a growing document with some answers. Call to request it. **Help improve it.**

Earn $50 credit toward any purchase when we add any description, code, or idea received from you.

SUGGESTED PRODUCTS FOR PRODUCTIVITY

- Full use of overlays, headers across 2.0 directories. **AKA ALIAS** $60.
- Recover erased or damaged files - use **Norton Utilities** $65 or **POWER** $139.
- Improve on DEBUG with **Codesmith 86** $135 many breakpoints / windows. Disasm.
- Analyze execution of any E X E with Profiler for $175.
- 200 C functions for DOS interface, PC screen, graphics, etc. **Greenleaf** $165.

RECENT DISCOVERIES HIGHLIGHTED

C Helper includes source in C for CP/M80, MSDOS for a DIFF, GREP, Flowcharter, XREF, C Beautifier and others. $125.

Note: all prices subject to change without notice. Mention this ad. Some prices are specials.

Call for a catalog, literature, comparisons, prices. ALL FORMATS.

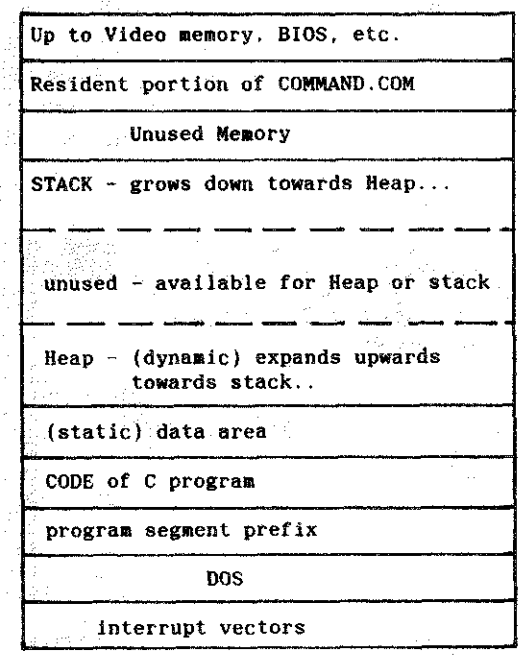
THE PROGRAMMER'S SHOP™
908-J Providence Highway, Dedham, MA 02026
617-461-0120, Mass: 800-442-8070
VISA 800-421-8006 MC

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

THE HEAP

A real Heap, on the other hand, consists of objects "purchased" from the operating system using functions like "alloc" which manage available memory. The Heap grows and contracts as requests are made to allocate or free up blocks. Since there is no knowledge at compile time of just how much storage will be needed (the number of requested bytes may be a computed number), there is no way to tell the linker to reserve any, so no attempt is made. Instead, the area just above static data (see fig. 1) is monitored by 'alloc' and friends. These functions maintain lists of pointers to memory - and make them available on request. In return for this service, the programmer should be kind enough to release them by using the function "free" (or something similar to it).

Figure 1 - Memory Usage in C



The "alloc" function establishes a memory address (pointer) to a region on the Heap which is unknown to the linker. A piece of C code to allocate and release memory might look like this:

```
char *mystring;
mystring = alloc( n );  /* n has been computed before here    */
dothis( mystring, whatever );   /* let ""dothis'' have control */
free( mystring );               /* free up the memory          */
```

The "mystring" pointer is passed to the "dothis" function, which does whatever it wants to with the memory space. Once the "free" function is called, the memory block pointed to by "mystring" is once again open for use.

ARGUMENT PASSING

When a function is called in C, arguments are pushed onto the stack.

Example 6 - Argument Passing

```
fx()
{
int a,b,c;
fy(a,b,c);
}
```

```
fx      proc    near
        push    bp
        sub     sp,6            ; allocate space for 3 integers
        mov     bp,sp           ; establish frame of reference
        push    [BP+4]          ; push "c"
        push    [BP+2]          ; push "b"
        push    [BP+0]          ; push "a"
        call    fy              ;  call the subordinate function
        mov     sp,bp
        add     sp,6            ; de-allocate auto variables
        pop     bp
        ret
fx      endp
```

In this case, "fx" is calling another function named "fy". What do you think "fy" will need if it plans to use the same "a,b,c" variables? Notice that all three integers are passed by value to the called function.

Example 7 - Function Receives Argument Values

```
fy( x,y,z )
int x,y,z;
{
x=1;    y=2;    z=3;
}
```

```
fy      proc    far
        push    bp
        mov     bp,sp                   ; no local variables..
        mov     word [BP+4],0001        ; set value of x
        mov     word [BP+6],0002        ;     value of y
        mov     word [BP+8],0003        ;           of z
        pop     bp
        ret
fy      endp
```

Oops — notice the offsets from BP in the last example. What are the first four bytes used for on the stack? (The words at BP+0 and BP+2 are not addressed by the function "fy") ? With small memory model functions, the first 2 bytes are occupied by the return address for the calling function (put there by the CALL instruction). The next two are used by the PUSH BP instruction in "fy" itself. If you are using one of the memory models which generate far calls, a total of six bytes is required instead of four.

When two or more functions are compiled into a program, the events become more obvious. Our next example illustrates this.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Example 8 - Argument Value Transfer to Called Function

```
fx()
{
int a,b,c;
a=1;    b=2;    c=3;
fy(a,b,c);
}
fy(x,y,z)
int x,y,z;
{
x=4;    y=5;    z=6;
}


fx      proc    near
        push    bp
        sub     sp,6            ; allocate room for a,b,c
        mov     bp,sp           ; establish addressability
        mov     word [BP+0],0001  ; a=1
        mov     word [BP+2],0002  ; b=2
        mov     word [BP+4],0003  ; c=3
        push    [BP+4]          ;  push "c" value
        push    [BP+2]          ;  push "b" value
        push    [BP+0]          ;  push "c" value
        call    fy              ;   and call "fy"
        mov     sp,bp           ; restore SP for exit
        add     sp,6            ;  after deallocating locals
        pop     bp              ; restore BP
        ret
fx      endp
fy      proc    near
        push    bp
        mov     bp,sp           ; ..no locals in this proc
        mov     word [BP+4],0001  ; set value of "x" on stack
        mov     word [BP+6],0002  ; same for "y"
        mov     word [BP+8],0003  ; ditto "z"
        pop     bp
        ret
fy      endp
```

The calling function saves only temporary values on the Stack. These throwaway values are then used by the receiving function. In "fx" the three variables are assigned specific values which are unknown to and cannot be changed by "fy". The stack grows when the arguments are passed to "fy". The arguments on the Stack may be altered by "fy", but "fx" will never know it.

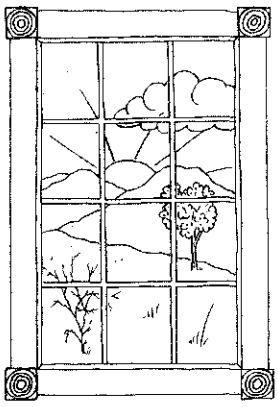
C does offer a simple mechanism by which the called function CAN alter the variables - by passing the actual address.

Example 9 - Argument Address Passed to Function

```
fx()
{
int a;
a=1;
fy( &a );
}

fy( x )
int *x;
{
*x = 2;
}


fx      proc    near
        push    bp
        sub     sp,2            ; allocate storage for a on stack
        mov     bp,sp           ; frame pointer
        mov     word [BP+0],0001  ; a=1
        lea     ax,[BP+00]      ; generate pointer to a
        push    ax              ; .. and pass that as argument
        call    fy
        mov     sp,bp
        add     sp,2
        pop     bp
        ret
fx      endp
fy      proc    near
        push    bp
        mov     bp,sp
        mov     si,[BP+4]       ; get address of argument
        mov     word [SI],0002  ;  and set value to 2
        pop     bp
        ret
fy      endp
```

In the above example, the "lea" instruction just loads the address of the argument into the accumulator. Then in "fy", this address is used in an index register so the integer's value can be altered. This illustrates an underlying principle about pointers in C: the little rascals were addresses all along !

NOW FOR THE IBM PC + COMPATIBLES



WINDOWS FOR C™

SCREEN MANAGEMENT TOOL FOR C PROGRAMMERS

COMPLETE WINDOW DISPLAY SYSTEM
- Unlimited windows and text files
- Horizontal and vertical scrolling
- Variable text margins
- ASCII file handling

SIMPLIFY • IMPROVE
- Menus
- Help files
- Data screens
- Editors

ALL DISPLAYS

ADVANCED FEATURES
- Instant screen changes
- No snow, no flicker
- Word wrap, auto scroll
- Overlay and restore windows
- Change attribute of selected text

C SOURCE MODULES FOR menus, cursor control, ASCII file display, text-mode bar graphs; + complete building block subroutines

Available For: Lattice C, C86, Microsoft C, DeSmet C (MS or PC DOS)

Introductory Special $119.95
($150 after May 31)
Demo disk & manual $30
(Applies toward purchase)

A PROFESSIONAL SOFTWARE TOOL FROM
CREATIVE SOLUTIONS
21 Elm Ave., Box P4, Richford, VT 05476

For Order or Information: 802-848-7738
Master Card & Visa Accepted
Shipping $2.50
VT residents add 4% tax

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

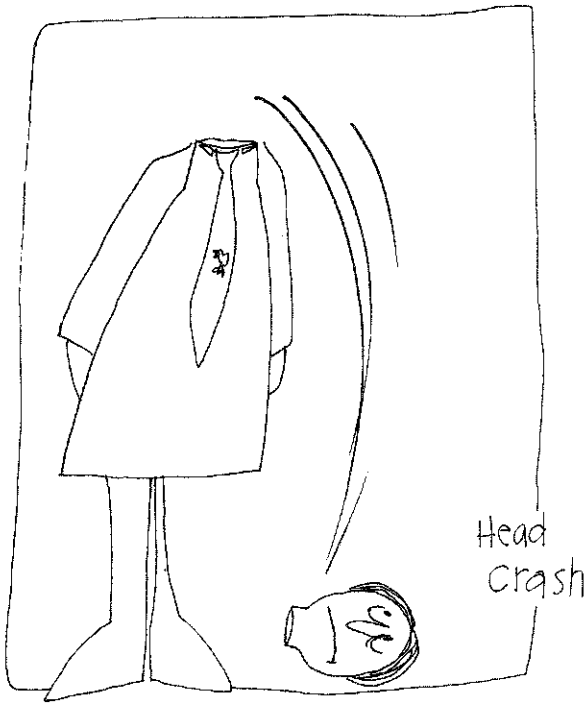
Example 9 defined "a" as an integer. Notice the difference in code generated if "a" is defined as a pointer to an integer.

Example 10 - Pointer as Argument

```
fx()
{
int *a;
*a = 1;         /* set initial value */
fy( a );
}

fy( x )
int *x;
{
*x = 2;
}
```

```
fx      proc    near
        push    bp
        sub     sp,2            ; reserve space for a
        mov     bp,sp
        mov     SI,[BP+0]       ; get address of a
        mov     word [SI],0001  ; *a = 1;
        push    SI              ; pass address of a to fy
        call    fy
        mov     sp,bp
        pop     bp
        ret
fx      endp
fy      proc    near
        push    bp
        mov     bp,sp           ; no locals in fy
        mov     SI,[BP+4]       ; first arg is address of a
        mov     word [SI],0002  ;  *a = 2;
        pop     bp
        ret
fy      endp
```

As a final example, let's examine a function taken from "The Greenleaf Functions," a library of general-purpose C functions designed to facilitate software development for the IBM PC. The "cprintf" function accepts arguments in the same manner as "printf," formats them into an internal buffer, then centers them on an indicated number of columns and prints them on "stdout."



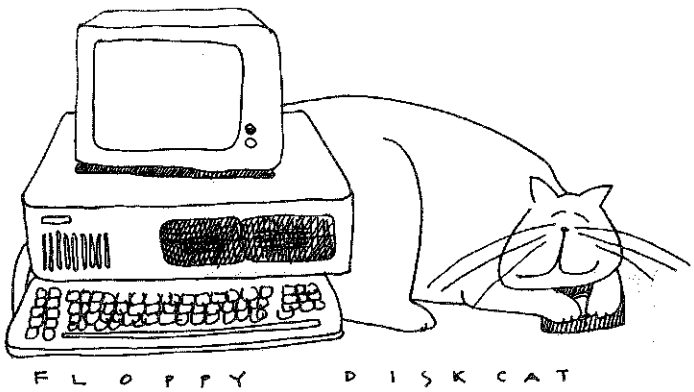
Example 11 - A Practical Function

```
/*  Center Print Formatted on stdout, compatible with redirection.
 *
 *  The Greenleaf Functions - Copyright (C) 1983 Greenleaf Software
 *
 *  cprintf( cols, format, args.. )
 *  char *str;          - string pointer
 *  int cols;           - justify or center on lesser of cols, MAXCOL
 *  char *format;       - see sprintf (same rules) - format specs
 *  -- args;           - see sprintf (same rules)
 *
 *  Returns:  NONE                          .
 */
#include "stdio.h"      /* pick up putchar macro        */
#define MAXCOLS 80      /* width of stdout screen       */
#define min(a,b) ((int)(a)<=(int)(b)?(int)(a):(int)(b))

void centers();         /* function to do centering     */
int sprintf();          /* see K&R or your compiler     */

void cprintf( cols, fmt, a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,
        a11,a12,a13,a14,a15,a16,a17,a18,a19,a20 )
int cols;
char *fmt;
unsigned a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,
        a11,a12,a13,a14,a15,a16,a17,a18,a19,a20;
{
int icols;
char buf[135];
  icols = min(cols,MAXCOLS);
  sprintf( buf, fmt,.a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,
          a11,a12,a13,a14,a15,a16,a17,a18,a19,a20 );
  cprints( icols, buf );
}

/*  cprints - Center and Print String on stdout
 */
void cprints( cols, str )
int cols;
char *str;
{
char out[135];
int icols;
  icols = min(cols,MAXCOLS);
  centers( out, str, icols );
  prints( out );
}

/*  prints - Print String on stdout (like "puts")
 */
void prints( str )
char *str;
{
char c, *p;
  p=str;
  while (c=*p++) putchar(c);
}
```

If we had tried to discern the actual number and type of arguments to be passed to "sprintf," we would have ended up lying to everyone including ourselves - so we just passed it some words. The "sprintf" function is intelligent enough to figure out what to do with them. The format string "fmt" is interpreted by "sprintf," not by "cprintf." Consequently "%d %f" is handled differently from "%f %d" but uses the same number of words on the stack.

```
cprintf proc    near
        push    bp
        sub     sp,8Ch          ; allocate space for buf (135)
        mov     bp,sp           ; establish frame pointer
        cmp     word [BP+90h],50  ;-- this is the code generated
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



FLOPPY DISK CAT

```
        jg      cprf2               ;-- by the "min" function (macro)
        mov     ax,[BP+90h]         ;--  use arg value
        jmp     cprf3               ;--
cprf2:  mov     ax,50h              ;--  use value from "MAXCOLS"
cprf3:  push    [BP+0BAh]       ; begin pushing arguments in
        push    [BP+0B8h]       ;  reverse order..
        .......
        push    [BP=090h]       ;   push a1
        push    [BP+092h]       ;   .. next to last .. fmt
        lea     bx,[BP+4]       ; address of buf
        push    bx              ; push that..
        mov     [BP+2],ax       ; save computed value of "icols"
        call    sprintf         ; get things formatted to buf
        mov     sp,bp           ; re-establish SP on return..
        lea     ax,[BP+4]       ; address of buffer
        push    ax              ;  as last arg to "cprints"
        push    [BP+2]          ; value of "icols" saved above
        call    cprints         ; now go center and print...
        mov     sp,bp           ; re-establish SP
        add     sp,08Ch         ; de-allocate autos and arg space
        pop     bp
        ret
cprintf endp

cprints proc    near            ; center and print string
        push    bp
        sub     sp,08Ch         ; allocate space for autos, temps
        mov     bp,sp
        cmp     word [BP+090h],050h     ;-- min does this
        jg      cprs4                   ;--
        mov     ax,[BP+090h]            ;--
        jmp     cprs5                   ;--
cprs4:  mov     ax,0050h                ;--
cprs5:  push    ax              ; save computed "icols" value
        push    [BP+092h]       ; ..and push it on stack as arg
        lea     bx,[BP+2]       ; address of "str"
        push    bx              ;  .. as second arg
        mov     [BP+08Ah],ax    ; "push" value of "icols"
        call    centers         ; get string centered.
        mov     sp,bp           ; assure SP integrity here
        lea     ax,[BP+2]       ; get address of buffer
        push    ax              ;  pass as argument
        call    prints          ;   to "prints"
        mov     sp,bp           ; restore integrity of SP
        add     sp,08Ch         ; de-allocate local storage
        pop     bp
        ret
cprints endp
prints  proc    near            ; print a string
        push    bp              ;..we simplify it here a bit..
        sub     sp,8
        mov     bp,sp
        mov     ax,[BP+0Ch]     ; address of "str"
        mov     [BP+6],ax       ; to "p" - local pointer
prns1:  mov     si,[BP+6]       ; get it again as index value
        inc     word [BP+6]     ;  increment pointer
        mov     al,[SI]         ; get character:  c=*p
        mov     [BP+4],al       ; store on stack - temporary
        test    al,al           ; is it zero (end-of-string) ?,
        jz      prns2           ;  if so, hop out
        ...                     ;----
        ...                     ; call to putchar is not so simple
        ...                     ;----
        mov     sp,bp           ; re-establish SP
        jmp     prns1           ; loop to do other characters
prns2:  add     sp,8            ; de-allocate locals
        pop     bp
        ret                     ; done !
prints  endp
```

Note that "cprintf" and relatives allocate temporary fixed length buffers on the stack rather than use the Heap via "alloc." This is a case of dealer's choice — a trade-off in favor of speed and simplicity. On the other hand, source code is supplied for "The Greenleaf Functions," so that users may change them if desired.

We have looked at stack usage, argument passing, and even located the Heap. We know enough about pointers already to live dangerously. Next time we'll explore structures and unions as arguments, learn more about pointers, and examine C's numerous data types. Who knows — we may find that K & R are real people after all.

References: (1) "The C Programming Language," Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall Inc., 1978

(2) "The Greenleaf Functions, User Manual," Greenleaf Software Inc., 1983.

Don Killen is president of Greenleaf Software Inc. and author of "The Greenleaf Functions."

INTERNATIONAL PC OWNERS

IPCO

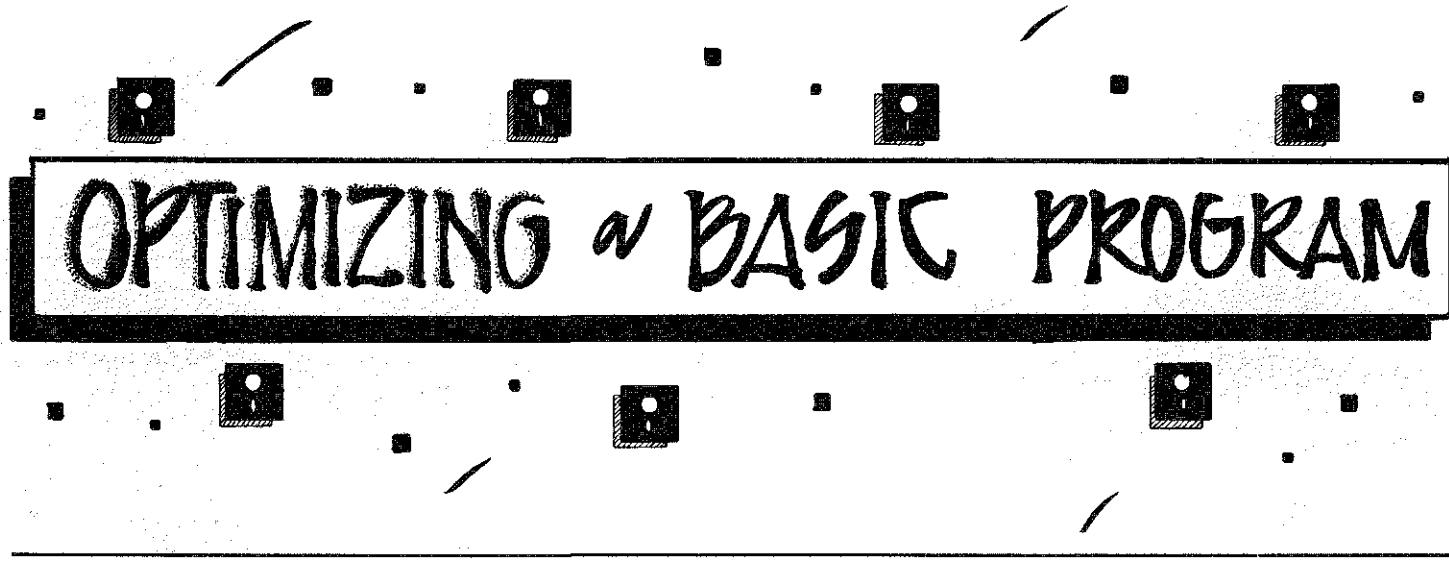
"The international user group for the PC"

- **Software library of 175+ programs @ $3 each**
- **Datacount® General Ledger/Manual $35**
- **Volkswriter® $125**
- **Bimonthly newsletter 40+ pages**
- **Worldwide information exchange**
- **"Freeware"**

FOR FREE CATALOG WRITE:

INTERNATIONAL PC OWNERS, INC.
ATTN: FREE CATALOG
P.O. BOX 10426
PITTSBURGH, PA 15234

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



OPTIMIZING BASIC COMPILER SOURCE CODE

by Wat Buchanon, President, The FreeSoft Company

Once a software application has been designed, the programmer is faced with an immediate decision: what language should the programs be written in? The fast execution speed of low-level languages like assembler and C is highly desirable, but the length of time necessary to write, assemble, and debug them is a definite drawback. One step up from the low-level languages are the mid-level compiled ones like Pascal and FORTRAN. Although the software development time is certainly less at this level, the compilation and linking phases necessitated by debugging can be quite time consuming. Finally, at the top of the heap, we find the high-level, interpreted languages like BASIC. Unfortunately, these languages have the slowest execution time for most projects and offer little, if any, source code security.

The IBM BASIC Compiler, in my opinion, comes closest to having the best of all three worlds. You get the development and debugging advantage of an interpreter, execution speed that approaches (and in some cases surpasses) mid-level languages, and the ability to access low-level functions with very few modifications. The source code versions for the interpreter and the compiler are very close. It has been my experience that a good 80 percent of the programs we write can be satisfactorily done in compiled BASIC. For those routines that inevitably require machine-language speed, such as display or diskette I/O, you need only create a library of callable routines and then attach them to the compiled program, using the LINK utility.

Like many programmers, I often find myself wanting to squeeze a little extra speed out of my applications, and save a few bytes in the process. One technique to achieve this is to use the line number optimizing switch with the BASIC Compiler.

The optimizing switch is one of the most neglected features of the BASIC Compiler, yet it offers two clear advantages to the programmer: less object code space requirements and faster program execution. The space benefit can make a big difference since the compiler places a limit on the size of your program. One software author I know was able to save 25 percent of the source code space by using the optimizing switch with the BASIC Compiler, and I'm sure even better results are possible.

The optimizing switch is invoked by adding /N to the source or object filespec in the compiler. It tells the compiler that line numbers will only be used by lines that need them. Let's look at an example:

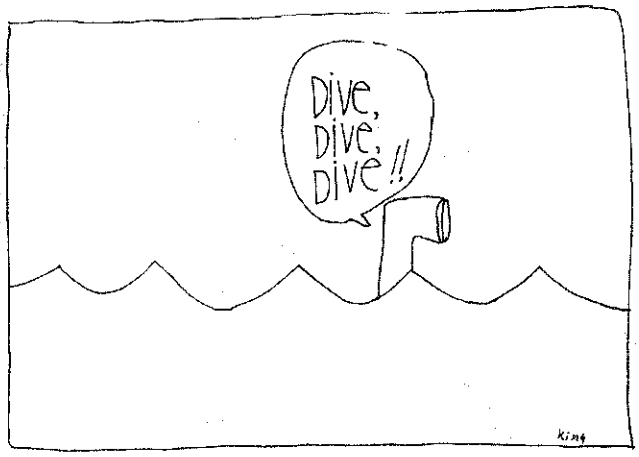
```
10 REM ** This is a sample program
20 PRINT "HELLO":GOTO 40
30 REM ** The word HELLO was printed
40 PRINT "END OF PROGRAM"
50 END
```

If you look closely at the program above, you'll see that only line 40 is really necessary. If the program was optimized for the compiler, it would look like:

```
   REM ** This is a sample program
   PRINT ""HELLO'':GOTO 40
   REM ** The the word HELLO was printed
40 PRINT ""END OF PROGRAM''
   END
```

The savings of such a compilation would be 8 bytes, since 2 bytes are used for each line number. This savings can add up

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



S U B R O U T I N E

fast, especially with large programs that have many unreferenced line numbers. The problem is that the BASIC interpreter is unable to load or execute line number optimized programs. Going in with a word processor and removing unnecessary line numbers once the program has been debugged is one solution, but you are likely to spend a great deal of time and introduce many errors in the process. What is really needed is a utility to automatically line number optimize a debugged BASIC program before it is submitted to the compiler, and such a utility is provided with this article.

Ultra-Optimize is one of the programs included with the Ultra-Utility package from The FreeSoft Company (see Programmer's Journal Vol. 2 No. 1). Below is the source code for this program so you can study its function. The program has only three inputs: the filespec of the program to optimize (your BASIC program), the filespec of the optimized program, and whether or not you wish the line numbers of the source program to be displayed while the program is optimizing. I use an identical name for the optimized file, but with the extension .CMP to remind me that the optimized file is only for the compiler. The resulting optimized program should operate like the unoptimized version in all respects except speed.

The program, as it is listed below in interpretive form, will run rather slowly. I recommend that you optimize it and then compile it for maximum execution speed. Also, if you use error-trapping routines in your program that display line numbers, remember to print out a copy of the optimized source code, since the line number reported will point to the 'block' containing the error, rather than to the individual line in your unoptimized version.

Finally, I would not be willing to enter into any barroom arguments over the programming style of the listing on the next two pages. It was written in a freehanded fashion, and would probably cause a twinge of nausea in many structured programming gurus. However, I make no apologies since the program works, took little time to develop, and serves my needs well. This is why I use the BASIC Compiler to begin with, being something of a pragmatist. Given the fact that the meat of Ultra-Optimize is string handling, I don't suppose a conversion to assembly language would be an insurmountable task, should one of you be so moved. Ultra-Optimize will theoretically handle BASIC programs of up to 10000 program lines, as specified in the dimension array in Line 20. It operates in two phases. In the first phase, each program line is scanned for references to other program lines. Typical commands that do this are the GOTO and GOSUB commands, although several other exist. If another program line is referenced, it is saved in the line number array. The second phase actually extracts the line numbers from each line, if they are not referenced by other program lines. The program should work equally well with a color or monochrome monitor.

I welcome any suggestions or improvements to this tool, or to any other Ultra-Utility program. For a complete copy of the Ultra-Utilities, send one double- or two single-sided diskettes with a postage-paid, self-addressed return mailer to:

Wat Buchanon
The FreeSoft Company
P.O. Box 27608
St. Louis, MO 63146

C BUILDING BLOCKS

Save a year of development.
Over 600 functions.
Main-frame libraries on a micro.

□ **IBM PC* Building Blocks** $149
All DOS functions, printer, asynchronous port, video, directories. Over 200 functions.
□ **Mathematics** ..... $ 99
Logarithms, trigonometric functions, square root.
□ **Communications** $149
For Hayes Smartmodem†, xon/xoff, modem7 and xmodem.
□ **B-trees & Database Management** $149
Variable length records, lists, duplicate keys.
□ **Advanced Building Blocks*** ..... $ 99
Julian Date, field input, graphic fill, pie, text windows.
□ **Archive Utility** ..... $ 49
Many files to one file. View/print while in archive.

Credit Cards Accepted. Single User License.
Multiuser licenses available. ($7.00 handling/Mass. add 5%)

NOVUM ORGANUM

29 Egerton Road, Arlington, MA 02174 Tel: (617) 641-1650

(TM) IBM* (TM) Hayes Microcomputer Products† *Require PC Building Blocks.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Programmers:
Rebuild Your Program Automatically By Typing 4 Letters:

POLYMAKE™

The
Intelligent Program Builder & Maintenance Tool
For MS-DOS

PolyMake is an enhanced version of the UNIX Make program. When a change is made to any module of a software system, PolyMake can automatically reconstruct files dependent upon the changed file.

Maintain Up-To-Date Files

PolyMake will invoke your compiler, assembler, linker, librarian, or do whatever is necessary to bring all dependent files up to date. PolyMake compares the date and time of all relevant files and uses internal rules to rebuild a program or complete software system. These rules can be modified and expanded by the user. All this will significantly reduce development time, prevent bugs, manage large software projects, and perform custom installations of programs or systems.

Experience New Freedom

PolyMake frees you from the need to remember which files depend on others and which files have been modified. PolyMake remembers the exact sequence of operations necessary to make a new version of a program.

Run UNIX Make-Files

Make-files developed on a mainframe can run on the IBM PC. PolyMake runs UNIX Make-files directly and uses the same UNIX Make-file syntax.

Bonus Discount

PolyMake is from the designers of PolyLibrarian, the $99 Object Module Librarian. You can order both PolyMake and PolyLibrarian for $160, a $38 discount.

Requires 128K RAM MS-DOS 1.1 or 2.0

VISA/MC ORDERS CALL
1-800-547-4000
ask for Dept. 306

$99 includes Comprehensive Manual with Tutorial (Include $1.50 shipping)

OR SEND CHECK, P.O.'s TO:

POLYTRON™

DS-3-306
P.O. Box 787
Hillsboro, OR 97123

Program Listing: Ultra-Optimize

```
1' Copyright (c) 1984 by Wat Buchanon & The FreeSoft Company
        All rights reserved
5' U-OPT.BAS, a BASIC program line optimizer
10 KEY OFF
20 CLS:DIM LN(10000):NUM$=""0123456789'':ON ERROR GOTO 440:
        PRINT""---> Ultra-Optimize <--- Volume #4 of 5 of The Ultra-Utilities''
30 KV=1:PRINT:PRINT
40 INPUT ""Enter filespec to optimize: '';F$
50 INPUT ""Enter optimized filespec '';OP$
60 PRINT""Display line numbers while working (Y/N)?''
70 DISP$=INKEY$:IF DISP$<>""Y'' AND DISP$<>""y'' AND DISP$<>""N''
        AND DISP$<>""n'' GOTO 70
80 OPEN ""i'',1,F$:PTR=1:LINE INPUT#1,A$:CLOSE:FOR Z=1 TO LEN(A$):
        IF MID$(A$,Z,1)<>"" '' THEN NEXT Z ELSE LNUM$=LEFT$(A$,Z-1):
        LNUM=VAL(LNUM$):LN(PTR)=LNUM:PTR=PTR+1:OPEN ""i'',1,F$
90 CLS:GOSUB 480
100 LINE INPUT#1,A$:FOR Z=1 TO LEN(A$):IF MID$(A$,Z,1)<>"" '' THEN
        NEXT Z ELSE LNUM$=LEFT$(A$,Z-1):LOCATE 5,1:GOSUB 590
110 QUOTE=-1:FOR Z=1 TO LEN(A$)-3:Z1$=MID$(A$,Z,1):Z3$=MID$(A$,Z,3):
        Z4$=MID$(A$,Z,4):Z5$=MID$(A$,Z,5):Z6$=MID$(A$,Z,6):Z7$=MID$(A$,Z,7)
120 IF Z1$=CHR$(34) THEN QUOTE=QUOTE*-1:GOTO 190
130 IF (Z1$=""'' OR Z3$=""REM'') AND QUOTE=-1 GOTO 200
140 IF (Z4$=""ELSE'' OR Z4$=""THEN'') AND QUOTE=-1 THEN B=5:GOSUB 210
150 IF Z4$=""GOTO'' AND QUOTE=-1 THEN B=5:GOSUB 340
160 IF Z5$=""GOSUB'' AND QUOTE=-1 THEN B=6:GOSUB 340
170 IF Z6$=""RESUME'' AND QUOTE=-1 THEN B=7:GOSUB 210
180 IF Z7$=""RESTORE'' AND QUOTE=-1 THEN B=8:GOSUB 210
190 NEXT Z
200 IF EOF(1) GOTO 250 ELSE GOTO 100
210 Q=Z+B:Q$=MID$(A$,Q,1):IF INSTR(NUM$,Q$)=0 THEN RETURN
220 ST=Q
230 Q=Q+1:Q$=MID$(A$,Q,1):IF Q$<>""'' AND INSTR(NUM$,Q$)<>0 GOTO 230
240 LGTH=Q-ST:LNUM$=MID$(A$,ST,LGTH):LNUM=VAL(LNUM$):LN(PTR)=LNUM:
        PTR=PTR+1:RETURN
250 CLOSE:GOSUB 560
260 OPEN ""I'',1,F$:OPEN ""O'',2,OP$
270 LINE INPUT#1,A$:FOR Z=1 TO LEN(A$):IF MID$(A$,Z,1)<>"" '' THEN
        NEXT Z ELSE LNUM$=LEFT$(A$,Z-1):CMP=VAL(LNUM$)
280 GOSUB 610
290 FOR K=1 TO PTR-1:IF LN(K)=CMP GOTO 310 ELSE NEXT K
300 J=LEN(A$):K$=RIGHT$(A$,J-(Z-1)):A$=K$
310 PRINT#2,A$
320 IF EOF(1) GOTO 330 ELSE GOTO 270
330 CLOSE:CLS:END
340 Q=Z+B:Q$=MID$(A$,Q,1):IF INSTR(NUM$,Q$)=0 THEN RETURN
350 ST=Q
360 Q=Q+1:Q$=MID$(A$,Q,1):IF Q$<>""'' AND INSTR(NUM$,Q$)<>0 GOTO 360
370 LGTH=Q-ST:LNUM$=MID$(A$,ST,LGTH):LNUM=VAL(LNUM$):LN(PTR)=LNUM:PTR=PTR+1
380 IF Q$<>"",'' THEN RETURN
390 Q=Q+1:Q$=MID$(A$,Q,1):IF Q$=""'' THEN RETURN ELSE IF INSTR(NUM$,Q$)=0
        GOTO 430
400 GOTO 350
410 R=PTR+1
420 IF Q$<>"",'' THEN RETURN
430 Q=Q+1:Q$=MID$(A$,Q,1):IF Q$="" THEN RETURN ELSE
        IF INSTR(NUM$,Q$)=0 GOTO 430
440 RESUME 450
450 CLS:PRINT""Unrecoverable Error in Line #'';ERL;"" OF ULTRA-OPTIMIZE''
460 PRINT""Consult accompanying article for operating instructions!''
470 END
480 CLS:TI$=""Ultra-Optimize'':B9=35:LOCATE 1,B9:PRINT TI$;
490 ST$=STRING$(40,205):DN$=CHR$(186)
500 LOCATE 3,22:PRINT ST$;:LOCATE 12,22:PRINT ST$;:FOR Z=4 TO 11:
        LOCATE Z,21:PRINT DN$;:LOCATE Z,62:PRINT DN$;:LOCATE Z+13,21:
        PRINT DN$;:LOCATE Z+13,62:PRINT DN$;:NEXT Z
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
510 LOCATE 3,21:PRINT CHR$(201);:LOCATE 3,62:PRINT CHR$(187);:
      LOCATE 12,21:PRINT CHR$(200);:LOCATE 12,62:PRINT CHR$(188);
520 LOCATE 16,22:PRINT ST$;:LOCATE 25,22:PRINT ST$;
530 LOCATE 16,21:PRINT CHR$(201);:LOCATE 16,62:PRINT CHR$(187);:
      LOCATE 25,21:PRINT CHR$(200);:LOCATE 25,62:PRINT CHR$(188);
540 LOCATE 4,25:PRINT""Pass One: '';:COLOR 31,0:PRINT""ACTIVE'';:COLOR 7,0:
      LOCATE 6,25:PRINT""Analyze lines for references'';:LOCATE 7,25:
      PRINT""to other BASIC lines...'';
550 RETURN
560 LOCATE 4,35:PRINT""      '';:LOCATE 17,25:PRINT""Pass Two: '';:
      COLOR 31,0:PRINT"ACTIVE";:COLOR 7,0:LOCATE 19,25:
      PRINT""Write optimized lines to new file...'';
570 LOCATE 9,25:PRINT""                          '';
580 RETURN
590 GOSUB 630:IF DISP$=""Y"'OR DISP$=""y'' THEN LOCATE 9,25:
      PRINT""Analyzing line #'';LNUM$;
600 RETURN
610 GOSUB 630:IF DISP$=""Y'' OR DISP$=""y'' THEN LOCATE 21,25:
      PRINT""Optimizing line #'';LNUM$;
620 RETURN
630 REM
640 LOCATE 1,34+KV:COLOR 7,0:PRINT MID$(TI$,KV,1);
650 KV=KV+1:IF KV>LEN(TI$) THEN KV=1
660 LOCATE 1,34+KV:COLOR 15,0:PRINT MID$(TI$,KV,1):COLOR 7,0
670 RETURN
```

□

Introducing the Creative Genius. . .

YOU. **You have the system . . . and now you have the means.** DataBurst™, a genuine full-screen processor and BASIC program generator, brings a System/34 interactive environment to your IBM® -PC. DataBurst™ provides screen design and program generator capabilities similar to SDA on the System/34. The unique DataBurst screen editor provides fast, easy screen design. Program independent screen formats reduce both development and maintenance time.

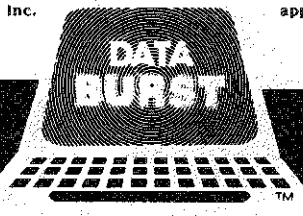
During execution of your program, DataBurst™ handles all of the screen and keyboard operations through one Assembly language module, requiring as little as 12K of memory. A true full-screen processor, DataBurst™ provides you with the power of System/34 SDA on your IBM® -PC.

DataBurst™ is available through your local computer retailer or direct from Key Solutions. To order directly, please send check, money order or purchase order for $225* to Key Solutions, Inc., P.O. Box 2297, Santa Clara, CA 95055. BASIC Compiler Runtime Support is available for an additional $40.* (Please inquire about the release dates for additional language interfaces.)

IBM® -PC is a registered trademark of IBM Corporation.
DataBurst™ is a trademark of Key Solutions, Inc.

*In California add applicable sales tax.

The Design Tool for the Creative Programmer



KEY SOLUTIONS

DeSmet C

The fastest 8088 C Compiler available

FULL DEVELOPMENT PACKAGE
- C Compiler
- Assembler
- Linker and Librarian
- Full-Screen Editor
- Newsletter for bugs/updates

SYMBOLIC DEBUGGER
- Monitor and change variables by name using C expressions
- Multi-Screen support for debugging PC graphics and interactive systems
- Optionally display C source during execution
- Breakpoint by Function and Line #

COMPLETE IMPLEMENTATION
- **Both** 1.0 **and** 2.0 DOS support
- Everything in K&R (incl. STDIO)
- Intel assembler mnemonics
- **Both** 8087 **and** Software Floating Point

OUTSTANDING PERFORMANCE

Sieve Benchmark

COMPILE	4 Sec.	RAM —
	22 Sec.	FDISK
LINK	6 Sec.	RAM —
	34 Sec.	FDISK
RUN	12 Sec.	
SIZE	8192 bytes	

DeSmet C Development Package **$159**

To Order Specify:

Machine _____

OS ☐ MS-DOS ☐ CP/M-86

Disk ☐ 8" ☐ 5¼ SS ☐ 5¼ DS

CWARE CORPORATION

P.O. BOX 710097
San Jose, CA 95171-0097
(408) 736-6905

California residents add sales tax. Shipping: U.S. no charge, Canada add $5, elsewhere add $15. Checks must be on a US Bank and in US Dollars.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

ASSEMBLY LANGUAGE TUTORIAL

Part I

by Joshua Auerbach

This talk is for people who are just getting started with the PC MACRO Assembler. Maybe you are just contemplating doing some coding in assembler, maybe you have tried it with mixed success. If you are here to get aimed in the right direction, to get off to a good start with the assembler, then you have come for the right reason. I can't promise you'll get what you want, but I'll do my best. On the other hand, if you have already turned out some working assembler code, then this talk is likely to be on the elementary side for you. If you want to review a few basics and have nowhere else pressing to go, then by all means stay.

WHY LEARN ASSEMBLER?

The reasons for LEARNING assembler are not the same as the reasons for USING it in a particular application. But, we have to start with some of the reasons for using it, and then I think the reasons for learning it will become clear. First, let's dispose of a bad reason for using it. Don't use it just because you think it is going to execute faster. A particular sequence of ordinary bread-and-butter computations written in PASCAL, C, FORTRAN, or compiled BASIC can do the job just about as fast as the same algorithm coded in assembler. Of course, interpretive BASIC is slower, but if you have a BASIC application which runs too slow you probably want to try compiling it before you think too much about translating parts of it to another language.

On the other hand, high-level languages do tend to isolate you from the machine. That is both their strength and their weakness. Usually, when implemented on a micro, a high-level language provides an escape mechanism to the underlying operating system or to the bare machine. So, for example, BASIC has its PEEK and POKE. But, the route to the bare machine is often a circuitous one, leading to tricky programming which is hard to follow.

For those of us working on PC's connected to SHARE-class mainframes, we are generally concerned with three interfaces: the keyboard, the screen, and the communication line or lines. All three of these entities raise machine dependent issues which are imperfectly addressed by the underlying operating system or by high level languages.

Sometimes, the system or the language does too little for you. For example, with the asynch adapter, the system provides no interrupt handler, no buffer, and no flow control. The application is stuck with the responsibility for monitoring that port and not missing any characters, then deciding what to do with all errors. BASIC does a reasonable job on some of this, but that is only BASIC. Most other languages do less. Sometimes, the system may do too much for you. System support for the keyboard is an example. At the hardware level, all 83 keys on the keyboard send unique codes when they are pressed, held down, and released. But someone has decided that certain keys, like Num Lock and Scroll Lock, are going to do certain things before the application even sees them and can't therefore be used as ordinary keys.

Sometimes, the system does about the right amount of stuff but does it less efficiently than it should. System support for the screen is in this class. If you use only the official interface to the screen, you sometimes slow your application down unacceptably. I said before, don't use assembler just to speed things up, but there I was talking about mainline code, which generally can't be speeded up much by assembler coding. A critical system interface is a different matter: sometimes we may have to use assembler to bypass a hopelessly inefficient implementation. We don't want to do this if we can avoid it, but sometimes we can't. Assembly language code can overcome these deficiencies. In some cases, you can also overcome these deficiencies by judicious use of the escape valves which your high-level language provides. In BASIC, you can PEEK and POKE and INP and OUT your way around a great many issues. In many other languages you can issue system calls and interrupts and usually manage, one way or other, to modify system memory. Writing handlers to take real-time hardware interrupts from the keyboard or asynch port, though is still going to be a problem in most languages. Some languages claim to let you do it, but I have yet to see an acceptably clean implementation done that way. The real reason why assembler is better than 'tricky POKEs' for writing machine-dependent code, is the same reason why PASCAL is better than assembler for writing a payroll package: it is easier to maintain.

Let the high-level language do what it does best, but recognize that there are some things which are best done in assembler code. The assembler, unlike the tricky POKE, can make judicious use of equates, macros, labels, and appropriately placed comments to show what is really going on in this machine-dependent realm where it thrives.

So, there are times when it becomes appropriate to write in assembler; given that, if you are a responsible programmer or manager, you will want to be "assembler-literate" so you can decide when assembler code should be written.

What do I mean by "assembler-literate?" I don't just mean understanding the 8086 architecture; I think that, even if you

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

don't write much assembler code yourself, you ought to understand the actual process of turning out assembler code and the various ways to incorporate it into an application. You ought to be able to tell good assembler code from bad, and appropriate assembler code from inappropriate.

Steps to becoming ASSEMBLER-LITERATE

1. Learn the 8086 architecture and most of the instruction set. Learn what you need to know and ignore what you don't. Reading: The 8086 Primer by Stephen Morse, published by Hayden. You need to read only two chapters, the one on machine organization and the one on the instruction set.

2. Learn about a few simple DOS function calls. Know what services the operating system provides. If appropriate, learn a little about other systems too. It will aid portability later on. Reading: Appendices D and E of the PC DOS manual.

3. Learn enough about the MACRO assembler and the LINKer to write some simple things that really work. Here, too, the main thing is figuring out what you don't need to know. Whatever you do, don't study the sample programs distributed with the assembler unless you have nothing better!

4. At the same time as you are learning the assembler itself, you will need to learn a few tools and concepts to properly combine your assembler code with the other things you do. If you plan to call assembler subroutines from a high-level language, you will need to study the interface notes provided in your language manual. Usually, this forms an appendix of some sort. If you plan to package your assembler routines as .COM programs, you will need to learn to do this. You should also learn to use DEBUG.

5. Read the Technical Reference, but very selectively. The most important things to know are the header comments in the BIOS listing. Next, you will want to learn about the RS 232 port and maybe about the video adapters.

Notice that the key thing in all five phases is being selective. It is easy to conclude that there is too much to learn unless you can throw away what you don't need. Most of the rest of this talk is going to deal with this very important question of what you need and don't need to learn in each phase. In some cases, I will have to leave you to do almost all of the learning; in others, I will teach a few salient points — enough, I hope, to get you started. I hope you understand that all I can do in an hour is get you started on the way.

PHASE 1:

LEARN THE ARCHITECTURE AND INSTRUCTION SET

The Morse book might seem like a lot of book to buy for just two really important chapters; other books devote a lot more space to the instruction set and give you a big, beautiful reference page on each instruction. And some of the other things in the Morse book, although interesting, really aren't very vital and are covered too sketchily to be of any real help. The reason I like the Morse book is that you can just read it; it has a very conversational style, is very lucid, and it tells you what you really need to know. Nothing really gets belabored too much; you can gracefully forget the things you don't use. And I very much recommend READING Morse rather than studying it. Get the big picture at this point.

Now, you want to concentrate on those things which are worth fixing in memory. After you read Morse, you should relate what you have learned to this outline.

1. You want to fix in your mind the idea of the four segment registers CODE, DATA, STACK, and EXTRA. This part is pretty easy to grasp. The 8086 and the 8088 use 20-bit addresses for memory, meaning that they can address up to one megabyte of memory. But, the registers and the address fields in all the instructions are no more than 16 bits long. So, how to address all of that memory? The solution is to put together two 16 bit quantities like this:

```
calculation  SSSS0 - value in the relevant segment register SHL 4
depicted in   AAAA - apparent address from register or instruction
hexadecimal -------
             RRRRR - real address placed on address bus
```

In other words, anytime memory is accessed, your program will supply a 16 bit address. Another 16 bit address is acquired from a segment register, left shifted four bits (one nibble) and added to it to form the real address. You can control the values in the segment registers and thus access any part of memory you want. But the segment registers are specialized: one for code, one for most data accesses, one for the stack (which we'll mention again), and one "extra" for additional data accesses.

Most people, when they first learn about this addressing scheme, become obsessed with converting everything to real 20-bit addresses. After a while, though, you get used to thinking in segment/offset form. You tend to get your segment registers set up at the beginning of the program, change them as little as possible, and think just in terms of symbolic locations in your program, as with any assembly language.

EXAMPLE:

```
MOV  AX,DATASEG
MOV  DS,AX          ; Set value of Data segment
ASSUME DS:DATASEG ; Tell assembler DS is usable
MOV  AX,PLACE       ; Access storage symbolically by 16- bit address
```

In the above example, the assembler knows that no special issues are involved, because the machine generally uses the DS register to complete a normal data reference.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

If you had used ES instead of DS in the above example, the assembler would have known what to do, also. In front of the MOV instruction which accessed the location PLACE, it would have placed the ES segment prefix. This would tell the machine that ES should be used, instead of DS, to complete the address.

Some conventions make it especially easy to forget about segment registers. For example, any program of the COM type gets control with all four segment registers containing the same value. This program executes in a simplified 64K address space. You can go outside this address space if you want, but you don't have to.

2. You will want to learn what other registers are available and learn their personalities.

AX and DX are general purpose registers. They become special only when accessing machine and system interfaces.

CX is a general purpose register which is slightly specialized for counting.

BX is a general purpose register which is slightly specialized for forming base-displacement addresses.

AX-DX can be divided in half, forming AH, AL, BH, BL, CH, CL, DH, DL.

SI and DI are strictly 16 bit. They can be used to form indexed addresses (like BX) and they are also used to point to strings.

SP is hardly ever manipulated. It is there to provide a stack.

BP is a manipulable cousin to SP. Use it to access data which has been pushed onto the stack.

Most 16 bit operations are legal (even if unusual) when performed in SI, DI, SP, or BP.

3. You will want to learn the classifications of operations available WITHOUT getting hung up in the details of how 8086 opcodes are constructed.

8086 opcodes are complex. Fortunately, the assembler opcodes used to assemble them are simple. When you read a book like Morse, you will learn some things which are worth knowing but NOT worth dwelling on.

a. 8086 and 8088 instructions can be broken up into subfields and bits with names like R/M, MOD, S and W. These parts of the instruction modify the basic operation in such ways depending on whether it is 8 bit or 16 bit; if 16 bit, whether all 16 bits of the data are given; whether the instruction is register to register, register to memory, or memory to register; for operands which are registers, which register; for operands which are memory, what base and index registers should be used in finding the data.

b. Also, some instructions are actually represented by several different machine opcodes depending on whether they deal with immediate data or not, or on other issues. There are some expedited forms which assume that one of the arguments is the most commonly used operand, like AX in the case of arithmetic.

There is no point in memorizing any of this detail; just distill the bottom line, which is, what kinds of operand combinations EXIST in the instruction set and what kinds don't. If you ask the assembler to ADD two things and the two things are things for which there is a legal ADD instruction somewhere in the instruction set, the assembler will find the right instruction and fill in all the modifier fields for you. I guess if you memorized all the opcode construction rules you might have a crack at being able to disassemble hex dumps by eye, as you may have learned to do somewhat with 370 assembler. I submit to you that this feat, if ever mastered by anyone, would be in the same class as playing the "Minute Waltz" in a minute; a curiosity only. Here is the basic matrix you should remember:

```
Two operands:          One operand:

R <-- M                R
M <-- R                M
R <-- R                S *
RM <--I
RM <--S  *
S <-- RM  *

* -- data moving instructions (MOV, PUSH, POP) only
S -- segment register (CS, DS, ES, SS)
R -- ordinary register (AX, BX, CX, DX, SI, DI, BP, SP,
                        @olAH, AL, BH, BL, CH, CL, DH, DL)
M -- one of the following:
        @olpure address
        [BX]+offset
        [BP]+offset
        any of the above indexed by SI
        any of the first three indexed by DI
```

4. Of course, you want to learn the operations themselves. As I've suggested, you want to learn the op codes as the assembler presents them, not as the CPU machine language presents them. So, even though there are many MOV op codes, you don't need to learn them. Basically, here is the instruction set:

a. Ordinary two-operand instructions. These instructions perform an operation and leave the result in place of one of the operands. They are:

1) ADD and ADC — addition, with or without including a carry from a previous addition

2) SUB and SBB — subtraction, with or without including a borrow from a previous subtraction

3) CMP — compare. It is useful to think of this as a subtraction with the answer being thrown away and neither operand

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

actually changed.

4) AND, OR, XOR — typical boolean operations

5) TEST — like an AND, except the answer is thrown away and neither operand is changed.

6) MOV — move data from source to target

7) LDS, LES, LEA — some specialized forms of MOV with side effects

b. Ordinary one operand instructions. These can take any of the operand forms described above. Usually, they perform the operation and leave the result in the stated place:

1) INC — increment contents
2) DEC — decrement contents
3) NEG — twos complement
4) NOT — ones complement
5) PUSH — value goes on stack (operand location itself unchanged)
6) POP — value taken from stack, replaces current value

c. Now you touch on some instructions which do not follow the general operand rules but which require the use of certain registers. The important ones are:

1) The multiply and divide instructions

2) The "adjust" instructions which help in performing arithmetic on ASCII or packed decimal data

3) The shift and rotate instructions. These have a restriction on the second operand: it must be either the immediate value 1 or the contents of the CL register.

4) IN and OUT, which send or receive data from one of the 1024 hardware ports

5) CBW and CWD — convert byte to word or word to double word by sign extension

d. Flow of control instructions. These deserve study in themselves, and we will discuss them a little more. They include:

1) CALL, RET — call and return

2) INT, IRET — interrupt and return-from-interrupt

3) JMP — jump or "branch"

4) LOOP, LOOPNZ, LOOPZ — special (and useful) instructions which implement a counted loop similar to the 370 BCT instruction

5) Various conditional jump instructions

e. String instructions. These implement a limited storage-to-storage instruction subset and are quite powerful. All of them have the property that:

1) The source of data is described by the combination DS and SI.

2) The destination of data is described by the combination ES and DI.

3) As part of the operation, the SI and/or DI register(s) is(are) incremented or decremented so that the operation can be repeated.

They include:

```
1)  CMPSB/CMPSW -- compare byte or word
2)  LODSB/LODSW -- load byte or word into AL or AX
3)  STOSB/STOSW -- store byte or word from AL or AX
4)  MOVSB/MOVSW -- move byte or word
5)  SCASB/SCASW -- compare byte or word with contents of AL or AX
6)  REP/REPE/REPNE -- a prefix which can be combined with any of
    the above instructions to make them execute repeatedly across a
    string of data whose length is held in CX
```

f. Flag instructions: CLI, STI, CLD, STD, CLC, STC. These can set or clear the interrupt (enabled) direction (for string operations) or carry flags.

The addressing summary and the instruction summary given above mask a lot of annoying little exceptions. For example, you can't POP CS, and although the R <— M form of LES is legal, the M <— R form isn't, etc., etc. My advice is:

a. Go for the general rules. b. Don't try to memorize the exceptions. c. Rely on common sense and the assembler to teach you about exceptions over time. A lot of the exceptions cover things you wouldn't want to do anyway.

5. A few instructions are rich enough and useful enough to warrant careful study. Here are a few final study guidelines:

a. It is well worth the time learning to use the string instruction set effectively. Among the most useful are

```
REP MOVSB       ; moves a string
REP STOSB       ; initializes memory
REPNE SCASB     ; look up occurence of character in string
REPE CMPSB      ; compare two strings
```

b. Similarly, if you have never written for a stack machine before, you will need to exercise PUSH and POP and get very comfortable with them because they are going to be good friends. If you are used to the 370, with lots of general purpose registers, you may find yourself feeling cramped at first with many fewer registers and many instructions having register restrictions. But you have a hidden ally: you need a register and you don't want to throw away what's in it? Just

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PUSH it, and when you are done, POP it back. This can lead to abuse. Never have more than two "expedient" PUSHes in effect and never leave something PUSHed across a major header comment or for more than about 15 instructions. An exception is the saving and restoring of registers at entrance to and exit from a subroutine; here, if the subroutine is long, you should probably PUSH everything which the caller may need saved, whether you will use the register or not, and POP it in reverse order at the end.

Be aware that CALL and INT push return address information on the stack and RET and IRET pop it off. It is a good idea to become familiar with the structure of the stack.

c. In practice, to invoke system services you will use the INT instruction. It is quite possible to use this instruction effectively in a cookbook fashion without knowing precisely how it works.

d. The transfer of control instructions (CALL, RET, JMP) deserve careful study to avoid confusion. You will learn that these can be classified as follows:

1) All three have the capability of being either NEAR (CS register unchanged) or FAR (CS register changed).
2) JMPs and CALLs can be DIRECT (target is assembled into instruction) or INDIRECT (target fetched from memory or register).
3) If NEAR and DIRECT, a JMP can be SHORT (less than 128 bytes away) or LONG.

In general, the third issue is not worth worrying about. On a for ward jump which is clearly VERY short, you can tell the assembler it is short and save one byte of code:

JMP SHORT CLOSE BY

On a backward jump, the assembler can figure it out for you. On a forward jump of dubious length, let the assembler default to a LONG form; at worst you waste one byte.

Also let the assembler worry about how the target address is to be represented, in absolute form or relative form.

e. The conditional jump set is rather confusing when studied apart from the assembler, but you do need to get a feeling for it. The interactions of the sign, carry, and overflow flags can get your mind stuttering pretty fast if you worry about it too much. What it boils down to, though, is:

```
JZ        means what it says
JNZ       means what it says
JG reater means ""if the SIGNED difference is positive''
JA bove   means ""if the UNSIGNED difference is positive''
JL ess    means ""if the SIGNED difference is negative''
JB elow   means ""if the UNSIGNED difference is negative''
JC arry   assembles the same as JB; it's an aesthetic choice
```

You should understand that all conditional jumps are inherently DIRECT, NEAR, and "short"; the 'short"part means that they can't go more than 128 bytes in either direction. Again, this is something you could easily imagine to be more of a problem than it is. I follow this simple approach:

1) When taking an abnormal exit from a block of code, I always use an unconditional jump. Who knows how far you are going to end up jumping by the time the program is finished? For example, I wouldn't code this:

```
TEST    AL,IDIBIT       ; Is the idiot bit on?
JNZ     OYVEY           ; Yes. Go to general cleanup.
```

Rather, I would probably code this:

```
    TEST    AL,IDIBIT       ; Is the idiot bit on?
    JZ      NOIDIOCY        ; No. I am saved.
    JMP     OYVEY           ; Yes. What can we say...
NOIDIOCY:
```

The latter, of course, is a jump around a jump. Some would say it is evil, but I submit it is hard to avoid in this language.

2) Otherwise, within a block of code, I use conditional jumps freely. If the block eventually grows so long that the assembler starts complaining that my conditional jumps are too long, I:

a) Consider reorganizing the block, but

b) Also consider changing some conditional jumps to their opposite and use the "jump around a jump" approach as shown above.

Enough about specific instructions!

6. Finally, in order to use the assembler effectively, you need to know the default rules for which segment registers are used to complete addresses in which situations.

a. CS is used to complete an address which is the target of a NEAR DIRECT jump. On a NEAR INDIRECT jump, DS is used to fetch the address from memory, but then CS is used to complete the address thus fetched. On FAR jumps, of course, CS is itself altered. The instruction counter is always implicitly pointing in the code segment.

b. SS is used to complete an address if BP is used in its formation. Otherwise, DS is always used to complete a data address.

c. On the string instructions, the target is always formed from ES and DI. The source is normally formed from DS and SI. If there is a segment prefix, it overrides the source, not the target.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

I think the best way to learn about DOS internals is to read the technical appendices in the manual. These are not as complete as we might wish, but they really aren't bad; I certainly have learned a lot from them. What you don't learn from them you might eventually learn via judicious disassembly of parts of DOS, but that shouldn't really be necessary.

From reading the technical appendices, you learn that interrupts 20H through 27H are used to communicate with DOS. Mostly, you will use interrupt 21H, the DOS function manager.

The function manager implements a great many services. You request the individual services by means of a function code in the AH register. For example, by putting a 9 in the AH register and issuing interrupt 21H, you tell DOS to print a message on the console screen.

Usually, but by no means always, the DX register is used to pass data for the service being requested. For example, on the print message service just mentioned, you would put the 16 bit address of the message in the DX register. The DS register is also implicitly part of this argument, in keeping with the universal segmentation rules.

In understanding DOS functions, it is useful to understand some history and also some of the philosophy of MS-DOS with regard to portability. Generally, you will find, once you read the technical information on DOS and also the IBM technical reference, you will know more than one way to do almost anything. Which is best? For example, to do asynch adapter I/O, you can use the DOS calls (pretty incomplete), you can use BIOS, or you can go directly to the hardware. The same thing is true for most of the other primitive I/O (keyboard or screen), although DOS is more likely to give you added value in these areas. When it comes to file I/O, DOS itself offers more than one interface. For example, there are four calls which read data from a file.

The way to decide rationally among these alternatives is by understanding the tradeoffs of functionality versus portability. Three kinds of portability need to be considered: machine portability, operating system portability (for example, the ability to assemble and run code under CP/M 86), and DOS version portability (the ability of a program to run under older versions of DOS.

Most of the functions originally offered in DOS 1.0 were direct descendants of CP/M functions; there is even a compatibility interface so that programs which have been translated instruction for instruction from 8080 assembler to 8086 assembler might have a reasonable chance of running if they use only the core CP/M function set. Among the most generally useful in this original compatibility set are,

09 — print a full message on the screen
0A — get a console input line with full DOS editing
0F — open a file
10 — close a file (really needed only when writing)
11 — find first file matching a pattern
12 — find next file matching a pattern
13 — erase a file
16 — create a file
17 — rename a file
1A — set disk transfer address

The next set provide no function above what you can get with BIOS calls or more specialized DOS calls. However, they are preferable to BIOS calls when portability is an issue.
00 — terminate execution
01 — read keyboard character
02 — write screen character
03 — read COM port character
04 — write COM port character
05 — print a character
06 — read keyboard or write screen with no editing

The standard file I/O calls are inferior to the specialized DOS calls but have the advantage of making the program easier to port to CP/M, style systems. Thus they are worth mentioning:
14 — sequential read from file
15 — sequential write to file
21 — random read from file
22 — random write to file
23 — determine file size
24 — set random record

In addition to the CP/M compatible services, DOS also offers some specialized services which have been available in all releases of DOS. These include:
27 — multi-record random read
28 — multi-record random write
29 — parse filename
2A-2D — get and set date and time

All of the calls mentioned above which have anything to do with files make use of a data area called the "FILE CONTROL BLOCK" (FCB). The FCB is anywhere from 33 to 37 bytes long depending on how it is used. You are responsible for creating an FCB and filling in the first 12 bytes, which contain a drive code, a filename, and an extension. When you open the FCB, the system fills in the next 20 bytes, which includes a logical record length. The initial lrecl is always 128 bytes, to achieve CP/M compatibility. The system also provides other useful information such as the file size.

After you have opened the FCB, you can change the logical record length. If you do this, your program is no longer CP/M compatible, but that doesn't make it a bad thing to do. DOS documentation suggests you use a logical record length of one for maximum flexibility. This is usually a good recommendation.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

To perform actual I/O to a file, you eventually need to fill in byte 33 or possibly bytes 34-37 of the FCB. Here you supply information about the record you are interested in reading or writing. For the most part, this part of the interface is compatible with CP/M. In general, you do not need to (and should not) modify other parts of the FCB.

The FCB is pretty well described in appendix E of the DOS manual. DOS 2.0 has a whole new system of calls for managing files which does not require building an FCB at all. These calls are quite incompatible with CP/M and also mean that your program cannot run under older releases of DOS. However, these calls are very nice and easy to use. They have these characteristics:

1. To open, create, delete, or rename a file, you need only a character string representing its name.

2. The open and create calls return a 16 bit value which is simply placed in the BX register on subsequent calls to refer to the file.

3. There is not a separate call required to specify the data buffer.

4. Any number of bytes can be transferred on a single call; no data area must be manipulated to do this.

The "new" DOS calls also include comprehensive functions to manipulate the new chained directory structure and to allocate and free memory.

(The conclusion will follow in the next issue of Programmer's Journal.)

Joshua Auerbach developed this tutorial while working as a programmer for the Yale University Computing Center. He is now working for the IBM Research Division, Thomas J. Watson Research Staff Member, Yorktown Heights, NY.

This paper was part of the Integrated Personal Computers Project, Installation Code YU, Communications Group, Communications and Data Base Division, Session C316

The article was reprinted courtesy of SHARE, Inc. Copyright (C) 1984 by SHARE, Inc. Published by SHARE, Inc, a non-profit Delaware membership corporation, permission to reprint in whole or in part may be granted for educational or scientific purposes upon written application to the secretary of:
SHARE, Inc, 1 Illinois Center, 111 E. Wacker Dr., Chicago, Il. 60601
□

IBM PC programs wanted! Looking for state-of-the-art educational and personal productivity products. Contact:

PM INC.
Box 1526
Novato, CA 94947

Leading Educational Developer needs system designer and programmers for major new product. Contact:

Avant-Garde Creations
Box 30160
Eugene, OR 97403

Converting Apple line to IBM PC. Need programmers. Contact:

Mere Inc.
Box 5465
Eugene, OR 97405

Developing IBM authoring system for educational line. Write:

Universal Systems for Education, Inc.
14901 E. Hampden Avenue, Suite 250
Aurora, CO 80014

PROGRAMMERS' EXCHANGE

Introducing a new feature! PROGRAMMER'S JOURNAL announces a clearing center of current opportunities for IBM PC programmers who have programs to sell, companies who are looking for IBM PC programmers, and those who may want to exchange ideas with other professionals in the field.

Simply put your message in 25 words or less and include your address.

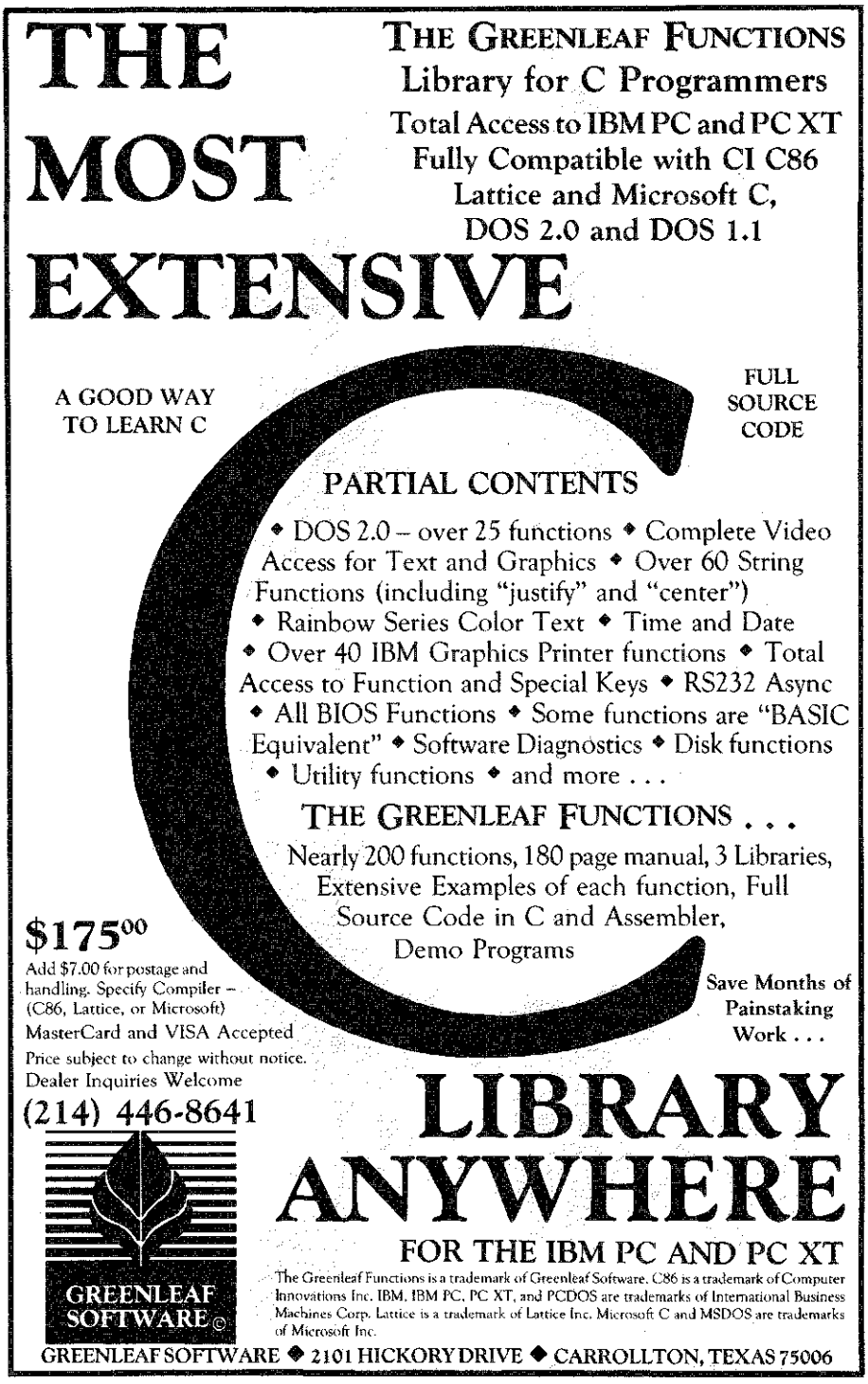
Cost is a modest $50.00 per issue to reach the leading people and companies in the IBM PC programming world.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

DATE, TIME AND FREE SPACE

from Ted A. Reuss

Here are a series of utility routines that should be quite useful in a number of circumstances for both C and Assembly programmers.


THE MOST EXTENSIVE
THE GREENLEAF FUNCTIONS
Library for C Programmers
Total Access to IBM PC and PC XT
Fully Compatible with CI C86
Lattice and Microsoft C,
DOS 2.0 and DOS 1.1
A GOOD WAY TO LEARN C
FULL SOURCE CODE
PARTIAL CONTENTS
◆ DOS 2.0 – over 25 functions ◆ Complete Video Access for Text and Graphics ◆ Over 60 String Functions (including "justify" and "center") ◆ Rainbow Series Color Text ◆ Time and Date ◆ Over 40 IBM Graphics Printer functions ◆ Total Access to Function and Special Keys ◆ RS232 Async ◆ All BIOS Functions ◆ Some functions are "BASIC Equivalent" ◆ Software Diagnostics ◆ Disk functions ◆ Utility functions ◆ and more . . .
THE GREENLEAF FUNCTIONS . . .
Nearly 200 functions, 180 page manual, 3 Libraries, Extensive Examples of each function, Full Source Code in C and Assembler, Demo Programs
$175.00
Add $7.00 for postage and handling. Specify Compiler – (C86, Lattice, or Microsoft)
MasterCard and VISA Accepted
Price subject to change without notice.
Dealer Inquiries Welcome
(214) 446-8641
Save Months of Painstaking Work . . .
LIBRARY ANYWHERE
FOR THE IBM PC AND PC XT
GREENLEAF SOFTWARE ©
The Greenleaf Functions is a trademark of Greenleaf Software. C86 is a trademark of Computer Innovations Inc. IBM, IBM PC, PC XT, and PCDOS are trademarks of International Business Machines Corp. Lattice is a trademark of Lattice Inc. Microsoft C and MSDOS are trademarks of Microsoft Inc.
GREENLEAF SOFTWARE ◆ 2101 HICKORY DRIVE ◆ CARROLLTON, TEXAS 75006

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
/*
    A series of Date and Time conversion functions
    written in Lattice/Microsoft C

    Ted A. Reuss c/o
    South Texas Software, Inc.
    4544 Post Oak Place, Ste 176
    Houston, TX  77027

    dtoa - date to alpha,
    ttoa - time to alpha,
    ttos - time to seconds
    atod - alpha to date

*/

#include <stdio.h>              /* Standard I/O definitions */
#include <brazos.h>
#include <ctype.h>              /* Character set conversion tables */

ushort atod(s)          /* cnvrt string date to packed_date */
char *s;
{
   ushort mm,dd,yy;

   if (sscanf(s,"%d/%d/%d",&mm,&dd,&yy) == 5)
   {
      if ( yy < 80 ) goto end;
      if ( yy >= 1980 ) yy-=1900;
      if ( mm < 1 || mm > 12 ) goto end;
      if ( dd < 1 || dd > 31 ) goto end;
      return (((yy-80)<<9) | (mm<<5) | dd);
   }
   if (sscanf(s,"%d-%d-%d",&mm,&dd,&yy) == 5)
   {
      if ( yy < 80 ) goto end;
      if ( yy >= 1980 ) yy-=1900;
      if ( mm < 1 || mm > 12 ) goto end;
      if ( dd < 1 || dd > 31 ) goto end;
      return (((yy-80)<<9) | (mm<<5) | dd);
   }
end:
   return(-1);
}

char *dtoa(d,s)         /* dtoa -- convert packed_date d to string s */
char s[];
int d;
{
   int i;
   itoa(i = d >> 5 & 0xf,s);
   if (i < 10) {
      s[1] = s[0];
      s[0] = '0';
   }
   s[2] = '/';
   itoa(i = d & 0x1f,&s[3]);
   if (i < 10) {
      s[4] = s[3];
      s[3] = '0';
   }
   s[5] = '/';
   itoa(i = (d >> 9 & 0x7f) + 80,&s[6]);
   if (i < 10) {
      s[7] = s[6];
      s[6] = '0';
   }
   return(s);
}

char *ttoa(t,s)         /* ttoa -- convert packed_time t to string s */
char s[];
int t;
{
   int i;
   itoa(i = t >> 11 & 0x1f,s);          /* hours */
   if (i < 10) {
      s[1] = s[0];
      s[0] = '0';
   }
   s[2] = ':';
   itoa(i = t >> 5 & 0x3f,&s[3]);       /* minutes */
   if (i < 10) {
      s[4] = s[3];
      s[3] = '0';
   }
   s[5] = ':';
   itoa(i = (t & 0x1f)*2,&s[6]);        /* seconds */
   if (i < 10) {
      s[7] = s[6];
      s[6] = '0';
   }
   return(s);
}

long ttos(pt)                /* ttos -- convert packed_time to seconds */
ushort pt;
{
   return ( (pt>>11&0x1f)*3600L + (pt>>5&0x3f)*60L + (pt&0x1f)*2L );
}

-------- brazos.h -------------

/** brazos.h -- Brazos system constants                          11.21.83
               Include in all brazos source files.
**/
#ifndef BRAZOS
#define BRAZOS 1
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
#define MAXLINE   128          /* Maximum input buffer size */
#define MAXNAME    64          /* Maximum size of file name with path */

/*===============   SCREEN BOUNDARY DEFINITIONS     ================*/
#define HEADNG_ROW  0          /* Heading line */
#define STATUS_ROW  1          /* Status messages */
#define PROMPT_ROW 22          /* Input prompts */
#define ERRMSG_ROW 23          /* Error messages */
#define HELP_ROW   24          /* Help prompts & info */

#include <syscons.h>           /* Typdefs and global constants */
#include <keymap.h>            /* Keyboard defs */
#include <ctype.h>             /* Character set conversion tables */
#endif


/** syscons.h  -- System constants for Lattice 'C' 10.5.83
**/
#ifndef FAIL
#define FAIL        1
#define SUCCESS     0
#define TRUE       (-1)
#define FALSE       0
#define NULL        0      /* null pointer value */
#define FKEY       (-1)    /* Function Key (see keymap.h) */
#define FOREVER     for (;;)
#define PAUSE       if (getch()=='\0') getch();

typedef char        byte;
typedef unsigned    ushort;
typedef int         void;
typedef int         (*PFI) ();  /* Pointer to function returning int     */
typedef long        (*PFL) ();  /*    "          "         "     long    */
typedef char *      (*PFC) ();  /*    "          "         "     char *  */

/*--- Data Types for getfld(), updfld(), etc. ---*/
#define STRING      0      /* Character String */
#define INTEGER     1      /* Integer */
#define REAL        2      /* Real or Float */
#define DATE        3      /* Date string  (mm/dd/yy) */
#define POS_INT     4      /* Positive Integer */
#define CHAR        5      /* A single character */
#define BYTE        6      /* A single byte */

/*------- Video Attributes for Mono-Chrome -------*/
#define NORMAL      0x07     /* 0000 0111 */
#define REVERSE     0x70     /* 0111 0000 */
#define UNDERLINE   0x01     /* 0000 0001 */
#define INTENSE     0x08     /* 0000 1000 */
#define BLINK       0x80     /* 1000 0000 */


ushort dateb(), timeb(), atod();
char  *getmem(), *itoa(), *dtoa(), *reverse(), *rmvcrlf(), *ttoa();
long   ttos(), freespc(), filesiz();
```

```
#define abs(x) ((x)<0?-(x):(x))          /* Absolute value */
#define max(a,b) ((a)>(b)?(a):(b))       /* Maximum value */
#define min(a,b) ((a)<=(b)?(a):(b))      /* Minimum value */
#endif


-------- keymap.h --------------

/** keymap.h - keyboard mapping   ( biased by 58 )          11.21.83
**/
#define FK1    (-1)
#define FK2    (-2)
#define FK3    (-3)
#define FK4    (-4)
#define FK5    (-5)
#define FK6    (-6)
#define FK7    (-7)
#define FK8    (-8)
#define FK9    (-9)
#define FK0    (-10)
#define HOME   (-13)                  /* home */
#define LNUP   (-14)                  /* up arrow */
#define PGUP   (-15)                  /* page up */
#define LEFT   (-17)                  /* left arrow */
#define RIGHT  (-19)                  /* right arrow */
#define END    (-21)                  /* end */
#define LNDN   (-22)                  /* down arrow */
#define PGDN   (-23)                  /* page down */
#define ESC 0x1b

#define HELP FK1         /* Function key assignments */
#define QUIT FK2
#define WNDW FK3
#define STAT FK4
#define ADD  FK5
#define DEL  FK6
#define UPD  FK7
#define CAN  FK8
#define PREV FK9
#define NEXT FK0

---------- DATE.ASM ------------


TITLE   DATE AND TIME ROTUINES FOR LATTICE 'c'
PAGE    64,132
PGROUP  GROUP   PROG
PROG    SEGMENT BYTE PUBLIC 'PROG'
        PUBLIC  dateb,timeb
        ASSUME  CS:PGROUP

;
;  d = dateb();  d gets packed date
;
;              <        yy       > < mm  > <  dd   >
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
;               15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
;
;       where:   yy is 0-119 (1980-2099)
;                mm is 1-12
;                dd is 1-31
;

dateb   PROC    NEAR
        MOV     AH,2AH          ;DOS GET DATE FUNCTION
        INT     21H
        XOR     AX,AX           ;AX = 0
        SUB     CX,1980         ;CX = YEAR - 1980
        SHL     CL,1            ;POSITION yy
        MOV     AH,CL           ;AH = yy
        OR      AL,DL           ;FOLD IN dd
        XOR     DL,DL           ;DL = 0
        MOV     CL,3            ;SHIFT COUNT FOR mm
        SHR     DX,CL           ;POSITION mm
        OR      AX,DX           ;FOLD IN mm
        RET

dateb   ENDP
;
;
;  t = timeb();  t gets packed time
;
;               <     hh     > <    mm    > <  xx   >
;               15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
;
;       where:   hh is 0-23
;                mm is 0-59
;                xx is 0-29 (two-second increments)
;

timeb   PROC    NEAR
        MOV     AH,2CH          ;DOS GET TIME FUNCTION
        INT     21H
        XOR     AX,AX           ;AX = 0
        MOV     AL,CL           ;AL = mm
        MOV     CL,5            ;SHIFT COUNT FOR mm
        SHL     AX,CL           ;POSITION mm
        MOV     CL,3            ;SHIFT COUNT FOR hh
        SHL     CH,CL           ;POSITION hh
        OR      AH,CH           ;FOLD IN hh
        SHR     DH,1            ;SECONDS/2
        OR      AL,DH           ;FOLD IN xx
        RET
timeb   ENDP

PROG    ENDS
        END
```

-------- FREESPC.ASM -----------------

```
title   DISK FREE SPACE ROUTINE FOR LATTICE 'C' DOS 2.0 Version
page    64,132
pgroup  group   prog
prog    segment byte public 'prog'
        public  FREESPC
        assume  cs:pgroup
;
; name          freespc -- get free space left on disk
;
; synopsis      lspc = freespc(drv)
;               long  lspc;     amount of free space
;               short drv;      drive number
;                               0 = default drive
;                               1 = A:
;                               2 = B:
;
; description   This function returns the free space left
;               on the specified drive.
;
; returns       lspc >=  0L if successful
;               lspc  = -1L if error
;
FREESPC proc    near
        push    bp
        mov     bp,sp
        mov     dx,[bp]+4       ;Get drive #
        mov     ah,30H
        int     21H             ;Get DOS version #
        cmp     al,2
        jge     fre10           ;If 2.0 or greater
        mov     ax,-1
        mov     dx,ax
        jmp     short fre20
fre10:
        mov     ah,36H
        int     21H             ;Get disk free space
        mul     bx              ;(sectors/cluster) * (free clusters)
        mov     dx,ax
        mul     cx              ; * (bytes/cluster)
fre20:
        mov     bx,ax           ;BX<- low size
        mov     ax,dx           ;AX<- high size
        mov     sp,bp
        pop     bp
        ret
FREESPC endp

prog    ends
□       end
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

FULL-SCREEN INPUT ROUTINES

by Ron Moak

Over time, every programmer develops a library of standard subroutines that, with little modification, can be plugged into a variety of programs. Once these generic routines have been thoroughly debugged, you can spend your time concentrating on new algorithms instead of rewriting old code. The input routine is usually the first generic routine produced. BASIC's INPUT function may be fine for some applications, but the "Redo From Start" message can wreck havoc on a carefully designed input screen.

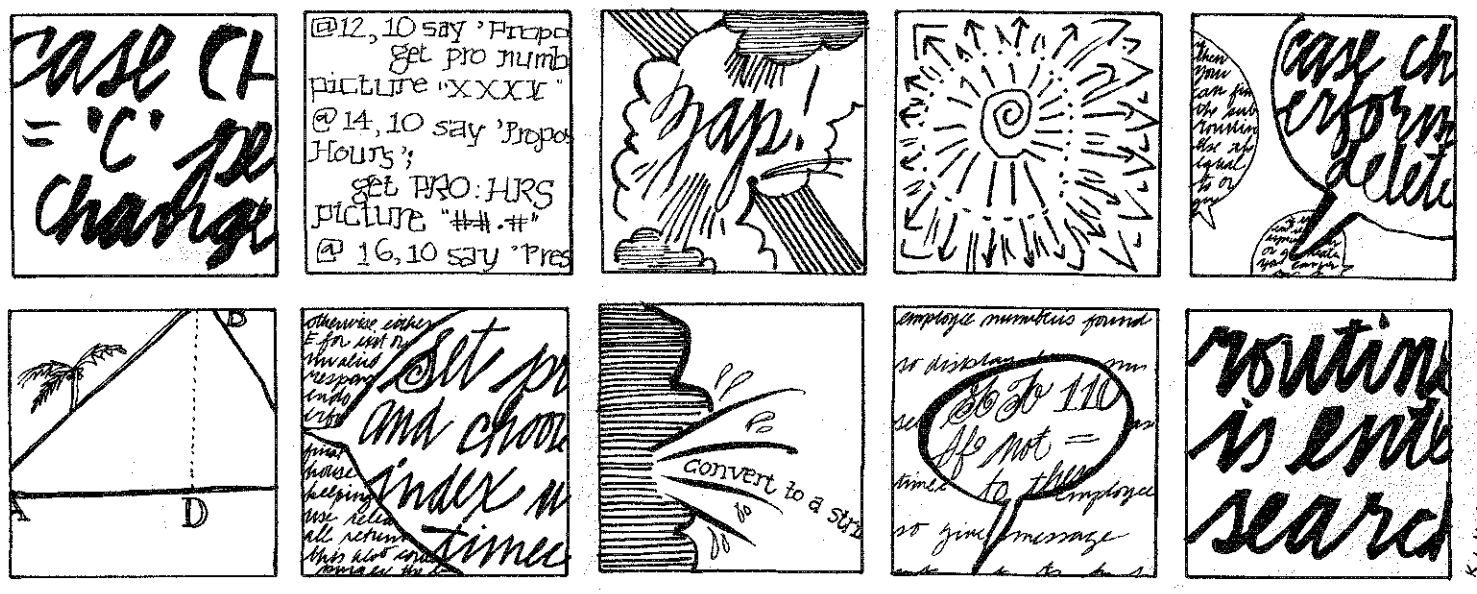
In general, most input routines are designed to receive the response from the keyboard and place it into a variable supplied by the calling function. The response is analyzed and, if acceptable, the next question is asked. Otherwise, the question is repeated with a possible error message. This sort of question-answer input is necessary if program direction is dependent on the response (as in the case of menus). But often a large amount of information is fed into the computer that does not affect program control.

For example, if you are entering information into a database, the standard question-and-answer method of entering data is inhibiting. In order to correct erroneous data, the operator must wait until all of the data for a record has been entered. Upon completion of the data input, the program must ask the operator if all of the information is correct. If anything in one of the fields is wrong, the user must indicate the incorrect item and then reenter the correct data. Only after a long series of questions and answers is the operator permitted to continue to the next record. A more usable approach would be to allow the operator to cursor around from field to field, entering data at random. When all of the data is entered, a separate keystroke would signal the end of input. The program could then read the data and perform any test for data integrity before continuing on to the next record.

The subroutines, at the end of this article, form the skeleton of an input module that allows the user to enter data in a full-screen editing mode. Before calling the routine, the programmer must first set up the screen with all of the prompts. Information about the types of responses allowed from the operator (type, field length, etc.) is placed in an array. Once this is done, program control is passed to the input routine.Movement around the screen is performed using the cursor keys. When all of the data is entered to the user's satisfaction, a single keystroke invokes another routine that is responsible for reading data into the program.

Aside from full-screen editing, the input routine allows input of data in either insert or overstrike mode. If you are, for instance, entering names and addresses, you may go back and enter characters in the middle of the field without retyping the entire field.

The key to the routine is the array called INGRID (for "input grid"), which contains the information about each variable requested from the screen. INGRID is a three-dimensional array; the first two elements decide the number and location of responses. For example, if your input data is ten rows by four columns, then the first two dimensions would be 10 and 4. If you call INGRID several times within the program using



PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

different inputs, then the array should be dimensioned according to the maximum number of inputs. The third dimension of INGRID has four elements. The first two elements contain the row and column of the input; the third holds the maximum number of characters allowed; and the fourth element contains the type of data (alphanumeric or numeric). The routine starting at line 100 gathers the initialization data for the INGRID array. Both the row, column, length and data type must be supplied. VARCROSS% and VARDOWN% are used to remember the width and depth of the grid.

Once the array INGRID is initialized, control is passed to subroutine 10200. The input routine is essentially one large while loop that traps every keystroke and responds accordingly. Control is passed back to the calling function when the ESCape key is pressed. When the arrow keys are pressed, the variables that keep track of the screen position (MROW%, MCOL%) are increased or decreased. Subroutine 10400 is used to display data being input with inverse video. The field size is indicated by the width of the inverse video area. When control is moved to another part of the screen, the old field is printed in normal video, then the active field is printed in inverse video.

As you have probably noticed, the date for each of the fields is not stored in temporary variables; instead, the screen buffer is used to hold all of the input. Any acceptable input is simply placed on the screen where it can be retrieved for later use.

Whenever a key is pressed, subroutine 10300 is activated. The contents of the screen buffer at the desired location are placed in the variable WORD$. Characters can then be added or deleted from the string. This requires a fair amount of calculation at each keystroke. Fortunately for us, the computer calculates faster than we can type. The real advantage to this method is that there is no required order to the keystrokes. A backspace in the middle of one field can be followed by a cursor instruction to move to another field.

Two subroutines, 10500 and 10600, are used for inserting and deleting characters from the screen. Routine 10500 has two modes of operation (set by the insert key): overstrike and insert. Under insert mode the characters in the field are spread apart at the cursor and the new character is inserted. If too many characters are inserted, characters on the left are lost. Overstrike simply replaces the character at the cursor with the new character. Subroutine 10600 is used to delete characters from the field by using either the backspace, or the DELete key. In either case, the characters in the field are compressed to prevent holes in the string.

The INGRID array uses a "1" in the type element to determine whether the field is alphanumeric, or a "0" if it is numeric only. The INSTR function is used to validate character input.

The last and most important function is to retrieve the newly entered information from the screen buffer and store it in program variables. Subroutine 10100 demonstrates a crude method of pulling data back into the program.The subroutine reads data from the screen buffer and places it in another array. With a little more coding, a more elaborate subroutine could be developed to test the the validity of the data as it is being transferred. If the test fails, control could be passed back to the input routine with an error message.

The INGRID routine is not generic and the elements must be reinitialized to reflect the desires of the calling function. Each type of data gathered (alpha or numeric) requires a unique version of the INGRID array.

```
'*********************************************************
'*
'*            General Full-Screen Input Routine
'*
'*                  (C)  Copyright 1984
'*
'*                    Ronald C. Moak
'*
'*********************************************************
    OPTION BASE 1
    DIM INGRID%(3,3,4), STORE.DATA%(3,3,4)

100 FOR I% = 1 TO 3                       ' 3 rows
       FOR J% = 1 TO 3                    ' up to 3 columns
          FOR N% = 1 TO 4                 ' 4 elements each
             READ STORE.DATA%(I%,J%,N%)   ' Data to matrix
          NEXT
       NEXT                               '
    NEXT                                  '

    DATA 5,17,20,0       ' row,col,length,character
    DATA 5,46,7,1        ' row,col,length,numeric
    DATA 5,46,7,1        ' dup as filler, no input
    DATA 7,20,25,0       ' row,col,length,charater
    DATA 7,20,25,0       ' dup as filler, no input
    DATA 7,20,25,0       ' dup as filler, no input
    DATA 9,17,20,0       ' row,col,length,character
    DATA 9,43,2,0        ' row,col,length,character
    DATA 9,53,5,1        ' row,col,length,numeric
'*********************************************************
'*
'*         CONSTANTS : Define known variables
'*
'*********************************************************
150 IROW% = 1: ICOL% = 2: ILEN% = 3: ITYPE% = 4
    TRUE% = 1: FALSE% = 0 : INSERT% = 82: DELE% = 83
    CURRIGHT% = 77: CURLEFT% = 75: CURUP% = 72: CURDOWN% = 8(
    CTRLCURRIGHT% = 116: CTRLCURLEFT% = 115: BACKSPACE% = 8
    ESC$ = CHR$(27)
'*********************************************************
'*
'*        Display data entry screen
'*
'*********************************************************
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
    LOCATE  3,31: PRINT "Input Client List";
    LOCATE  5,11: PRINT "Name:";
    LOCATE  5,39: PRINT "Phone:";
    LOCATE  7,11: PRINT "Address:";
    LOCATE  9,11: PRINT "City:";
    LOCATE  9,39: PRINT "St:";
    LOCATE  9,48: PRINT "Zip:";
    LOCATE 12,31: PRINT "Press <Esc> When Done";
    VARDOWN% = 3:  VARCROSS% = 3
    GOSUB 10000                         'Set Up Input Matrix

'*********************************************************
'*
'*     Call input routine & read data into program
'*********************************************************

    MROW% = 1: MCOL% = 1
    GOSUB 10200             'Get data
    GOSUB 10100             'Retrieve data to variables

'*********************************************************
'*       (Debugging purposes, redisplay data)
'*********************************************************
    LOCATE 15,11: PRINT "Name: "; NAME$;
    LOCATE 15,39: PRINT "Phone: "; PHONE!;
    LOCATE 17,11: PRINT "Address: "; ADDRESS$;
    LOCATE 19,11: PRINT "City: "; CITY$;
    LOCATE 19,39: PRINT "St: "; STATE$;
    LOCATE 19,48: PRINT "Zip: "; ZIP!;
    END
'*********************************************************
'*    Initialize array containing location, length & type
'*       of data to be input
'*********************************************************
10000 FOR MROW% = 1 TO 3
         FOR MCOL% = 1 TO 3
            FOR N% = 1 TO 4
               INGRID%(MROW%,MCOL%,N%) = STORE.DATA%
               (MROW%,MCOL%,N%)
            NEXT
         NEXT
      NEXT
      RETURN
'*********************************************************
'*
'*    get data from screen into variables
'*
'*********************************************************
10100 MROW% = 1: MCOL% = 1: GOSUB 10300
      NAME$ = WORD$
      MROW% = 1: MCOL% = 2: GOSUB 10300
      PHONE! = VAL(WORD$)
      MROW% = 2: MCOL% = 1: GOSUB 10300
      ADDRESS$ = WORD$
      MROW% = 3: MCOL% = 1: GOSUB 10300
      CITY$ = WORD$
      MROW% = 3: MCOL% = 2: GOSUB 10300
      STATE$ = WORD$
      MROW% = 3: MCOL% = 3: GOSUB 10300
      ZIP! = VAL(WORD$)
      RETURN

'*********************************************************
'*    Primary input routine
'*
'*    INPUTS  MROW%, MCOL%
'*********************************************************

10200 GET.INPUT% = TRUE%
      INSERT.ON% = 0
      LOCATE 25,70,1,6,7:
      PRINT "Replace";
      INV% = 1: GOSUB 10400

      WHILE GET.INPUT%
10210    INK$ = INKEY$: IF INK$ = "" THEN 10210
         IF LEN(INK$) = 1 THEN 10220
         INK% = ASC(RIGHT$(INK$,1))

         IF INK% = CTRLCURRIGHT% THEN_
            IF POS(0) < (INGRID%(MROW%,MCOL%,ICOL%) + _
               INGRID%(MROW%,MCOL%,ILEN%) - 1) THEN   _
                  LOCATE ,POS(0) + 1

         IF INK% = CTRLCURLEFT% THEN                  _
            IF POS(0) > INGRID%(MROW%,MCOL%,ICOL%) THEN _
               LOCATE ,POS(0) - 1

         IF INK% = CURUP% THEN          _'
            IF MROW% > 1 THEN           _' Test top margin
               GOSUB 10400             :_' Print normal vedio
               MROW% = MROW% - 1       :_' Update cursor
               INV% = 1: GOSUB 10400     ' Print inverse vedio

         IF INK% = CURDOWN% THEN        _'
            IF MROW% < VARDOWN% THEN    _' Test bottom margin
               GOSUB 10400             :_' Print normal vedio
               MROW% = MROW% + 1       :_' Update cursor
               INV% = 1: GOSUB 10400     ' Print inverse vedio

         IF INK% = CURRIGHT% THEN       _'
            IF MCOL% < VARCROSS% THEN   _' Test for right margin
               GOSUB 10400             :_' Print normal vedio
               MCOL% = MCOL% + 1       :_' Update cursor
               INV% = 1: GOSUB 10400     ' Print inverse vedio

         IF INK% = CURLEFT% THEN        _'
            IF MCOL% > 1 THEN           _' Test for left margin
               GOSUB 10400             :_' Print normal vedio

               MCOL% = MCOL% - 1       :_' Update cursor
               INV% = 1: GOSUB 10400     ' Print inverse vedio

         IF INK% = INSERT% THEN                    _'
            SY% = CSRLIN: SX% = POS(0)            :_' save cursor
            IF INSERT.ON% THEN                     _'
               INSERT.ON% = 0                     :_' Turn insert
               LOCATE 25,70: PRINT "Replace";     :_' flag off
               LOCATE SY%,SX%,1,6,7                _' restore cursor
            ELSE                                   _'
               INSERT.ON% = 1                     :_' Turn insert
               LOCATE 25,70: PRINT "Insert ";     :_' flag on
               LOCATE SY%,SX%,1,4,7                 ' restore cursor

         IF INK% = DELE% THEN_
            GOSUB 10600           ' Delete character under cursor

         GOTO 10230

10220    INK% = ASC(INK$)
         IF INK$ = ESC$ THEN       _' Completed input of data
            GOSUB 10400           :_' print normal vedio
            GET.INPUT% = FALSE%   :_' Clear loop flag
            GOTO 10230

         IF INK% = BACKSPACE% THEN                          _'
            IF POS(0) > INGRID%(MROW%,MCOL%,ICOL%) THEN     _'
               GOSUB 10600                                 :_'
               GOTO 10230
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
        IF INGRID%(MROW%,MCOL%,ITYPE%) THEN                    _'Test type
           IF INSTR(1,"1234567890-.",INK$) THEN                _'Test valid
              GOSUB 10500                                      _'Enter Num.
           ELSE                                                _'
              BEEP                                             _'Error
        ELSE                                                   _'
           IF INSTR(1,"ABCDEFGHIJKLMNOPQRSTUVWXYZ",INK$) OR    _'Test valid
              INSTR(1,"abcdefghijklmnopqrstuvwxyz",INK$) OR    _' characters
              INSTR(1,"1234567890.$ ",INK$) THEN               _'Enter Char
              GOSUB 10500                                      _'
           ELSE                                                _'
              BEEP
10230 WEND
      LOCATE 25,70: PRINT "       ";
      RETURN

'*******************************************************
'*
'*      Read Data off screen & into variable WORD$
'*
'*******************************************************
10300 WORD$ = ""
      FOR N% = 0 TO INGRID%(MROW%,MCOL%,ILEN%)-1
         WORD$ = WORD$ + CHR$(SCREEN(INGRID%(MROW%,MCOL%,IROW%),_

             INGRID%(MROW%,MCOL%,ICOL%) + N%))
      NEXT
      RETURN
'*******************************************************
'*       PRINT NORMAL  OR INVERSE VIDEO
'*
'*   Set INV%  to 1 for Inverse vedio
'*******************************************************
10400 GOSUB 10300                                  'Read screen
      SROW% = INGRID%(MROW%,MCOL%,IROW%)
      SCOL% = INGRID%(MROW%,MCOL%,ICOL%)
      LOCATE SROW%,SCOL%

      IF INV% THEN      _'
         COLOR 0,7     :_'
         PRINT WORD$; :_' Print inverse vedio
         COLOR 7,0     :_'
         INV% = 0       _'
      ELSE              _'
         COLOR 7,0     :_'
         PRINT WORD$;    ' Print normal vedio

      SROW% = INGRID%(MROW%,MCOL%,IROW%)
      SCOL% = INGRID%(MROW%,MCOL%,ICOL%)
      LOCATE SROW%,SCOL%
      RETURN
'*******************************************************
'*    Print characters to the screen in insert mode
'*               or overstrike mode
'*
'*   Set INSERT.ON% to 1 for insert mode
'*******************************************************
10500 SY% = CSRLIN: SX% = POS(0)                   'Old cursor
      SCOL% = INGRID%(MROW%,MCOL%,ICOL%)
      SROW% = INGRID%(MROW%,MCOL%,IROW%)
      LENG% = INGRID%(MROW%,MCOL%,ILEN%)
      END.WORD% = SCOL% + LENG% - 1
      IF INSERT.ON% THEN                                    _'
         GOSUB 10300                                       :_'Read screen
         IF POS(0) = END.WORD%  THEN                        _'Test for end
            LOCATE SROW%,SCOL%                             :_' of word$
            COLOR 0,7                                      :_'
            PRINT LEFT$(WORD$,LENG% - 1) + INK$;           :_'Add character
            COLOR 7,0                                      :_'
            LOCATE SY%,SX%                                  _'
         ELSE                                               _'
            CHAR.POS% = SX% - SCOL%                        :_'Cursor position
            WORD1$ = LEFT$(WORD$,CHAR.POS%)                :_'Left half word
            WORD2$ = MID$(WORD$,CHAR.POS% +1,LENG%)        :_'Right half word
            LOCATE SROW%,SCOL%                             :_'
            COLOR 0,7                                      :_'Add character to
            PRINT LEFT$(WORD1$+INK$+WORD2$,LENG%);         :_'middle of word.
            COLOR 7,0                                      :_'Excess characters
            LOCATE SY%,SX% + 1                              _'on right lost.
      ELSE                                                  _'
         IF POS(0) = END.WORD% THEN                         _'
            COLOR 0,7                                      :_'Overstrike the
            PRINT INK$;                                    :_'character at the
            COLOR 7,0                                      :_'end of the word.
            LOCATE ,POS(0) - 1                              _'move cursor back
         ELSE                                               _'
            COLOR 0,7                                      :_'Overstrike with
            PRINT INK$;                                    :_'new character.
            COLOR 7,0                                       'move cursor right
      RETURN
'*******************************************************
'*   Delete Characters from the Screen Input
'*******************************************************
10600 SCOL% = INGRID%(MROW%,MCOL%,ICOL%)
      SROW% = INGRID%(MROW%,MCOL%,IROW%)
      LENG% = INGRID%(MROW%,MCOL%,ILEN%)
      SY% = CSRLIN: SX% = POS(0)
      CHAR.POS% = SX% - SCOL%
      GOSUB 10300

      IF INK% = BACKSPACE% THEN                              _'
         WORD1$ = LEFT$(WORD$,CHAR.POS%-1)                  :_'
         WORD2$ = RIGHT$(WORD$,LENG% - CHAR.POS%)           :_'Delete character
         LOCATE SROW%,SCOL%                                 :_'to the right of
         SX% = SX% - 1                                      :_'cursor position.
         COLOR 0,7                                          :_'Join halves of
         PRINT LEFT$(WORD1$ + WORD2$,LENG%);                :_'word together.
         COLOR 7,0                                           _'
      ELSE                                                   _'
         WORD1$ = LEFT$(WORD$,CHAR.POS%)                    :_'Delete character
         WORD2$ = MID$(WORD$,CHAR.POS% + 2,LENG%-1)         :_'under current
         LOCATE SROW%,SCOL%                                 :_'cursor position.
         COLOR 0,7                                          :_'Join halves of
         PRINT LEFT$(WORD1$ + WORD2$,LENG%);                :_'word together.
         COLOR 7,0

      LOCATE SY%,SX%
      RETURN
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

MUSIC IN C

by Robert T. McCrory

BASICA provides extensive facilities for playing tones and music through the speaker on the IBM PC. Unfortunately, those of us who have taken "the road less traveled by" and who program in another language must create our own utilities to drown out the power supply fan.

For the last several months I have been programming in C, using the DeSmet C Development Package sold by C-Ware. The routines presented here will enable users of this compiler to play music through the IBM speaker; users of other packages will probably be able to adapt the routines with relatively little effort.

An assembly language module, SOUND.A forms the basis of all the routines. The routine tone(frequency,duration) plays a single tone based on two integer inputs: frequency in Hertz (cycles per second) and duration in hundredths of a second.

The tone routine is written using the DeSmet assembler - asm88. The logic for the routine is similar to the one in the ROM BIOS which is listed in the IBM Technical Reference Manual. Readers of Leo Scanlon's book, which has a sound routine also, should note that the listing contains some typographical errors.

It isn't necessary to go through the routine's logic blow by blow. The primary change you will need to use this routine with any C or Assembler package is in the interface to the calling routine. You will have to investigate how variables are passed to assembly language subroutines by C programs.

One should note that tone divides a constant by the frequency as part of its logic. Consequently, tone bombs and locks up your machine if you feed it a 0 frequency.

Interestingly enough, the routines that actually play the music are text-processing modules — they decode a string of symbols representing musical notes, determine frequency and duration, and call tone.

The symbols used in the text strings are as follows:

a through g and A through G — designate the pitch of the note. Upper or lower case letters may be used. The scale starts with A-440 but may be moved up or down as many octaves as desired.

+ and - — sharp and flat, respectively. These symbols move the pitch of the notes up or down a half tone. Multiple sharps and flats are permitted for each note, if desired.

> and < — move all subsequent notes up one octave or down one octave, respectively.

r and R — rests. The duration of the rest is determined in the same way as as the duration of a note (see below). In order to have silence during a rest, we call tone with a frequency of 32000; this is well beyond the range of the IBM speaker and human hearing. Remember that you cannot use 0 for the frequency, or tone will make your IBM catatonic.

0 through 9 — duration of the note. 1 designates a whole note, 2 a half note, 4 a quarter note, 16 a sixteenth note, and so on. The numbers and other symbols do not have to be together. The string a16 is a sixteenth note A, and so are 16A and 1a6. Because of the program logic, a 0 will cause a zero divide and lock up your machine.

. — increases the note duration by half. This is for dotted quarter notes and other exotica.

/ — separates one note from the next. Not needed after the last note in a string.

music.c contains two routines. playtune(tunestring,mm) takes a tune string and splits it up into individual note strings. These are passed with the metronome value mm to playnote(notestring,mm), which decodes the note string and plays the note by calling tone. mm is an integer which is the number of whole notes per second.

SOME LOONEY TUNES

I am a fan of the old Warner Bros. cartoons, particularly some of the better ones done by Chuck Jones, such as "What's Opera, Doc?" and "Duck Amuck." Therefore, I wanted to be

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

able to play two of the Looney Tunes theme songs, "Merrily We Roll Along" and "The Merry-Go-Round Broke Down."

These tunes are included in merrily.c as the modules merrily() and merry(). Each tune takes several lines; in DeSmet C, strings are limited to 80 characters, so you may need to split the string up into pieces. Fortunately, you can't hear the breaks.

In general, if you have some music you wish to play, the best way to proceed is to get the sheet music and transcribe the notes into a tune string. Experiment a bit with the metronome and you should have it.

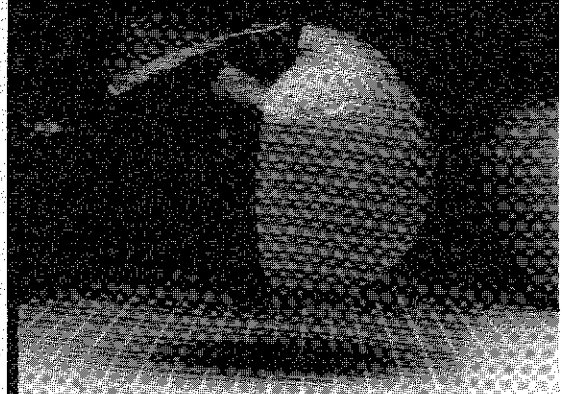
CONCLUSION

There is always room for improvement. These programs could be expanded to provide all the facilities that the new BASICA contains. I've accomplished what I set out to do, so I'm going to move on.

If you decide to make some changes or improvements, or if you think my rather straightforward code should be changed, let me know.

Robert T. McCrory
12064 S.E. 116th Court
Clackamas, OR 97015
□

NOW YOU CAN WORK IN C WITH ONLY BASIC KNOWLEDGE



Enjoy the Best of Both Worlds with BASIC_C™!

Write or Call Collect

SOURCE

C Source
12801 Frost Rd.
Kansas City, MO 64138

816-353-8808

Until now, IBM BASIC and C were about as far apart as apples and oranges — and if you wanted to move from BASIC to C, you had to go back to square one. Not anymore!

There's no denying that BASIC is a fine language: it has probably helped you learn a lot about programming computers. But now that you know how to program, isn't it time to take the next step — to a language that was designed to help you quickly develop software that is fast, compact, and portable?

BASIC_C can help you make the transition smoothly: it's a library of C functions providing BASIC's high level capabilities you've come to know so well. Functions for string handling, sequential and random i/o, graphics, and many more — all designed to look and work like BASIC.

You get the source code to BASIC_C: extensively documented, it provides many examples to help you learn how to write C functions, and if a function doesn't happen to suit your specific needs . . . change it!

C Source provides a complete development package: the BASIC_C Library, a full screen editor, and a choice of three C compilers. Buy it all, or just the pieces you need.

BASIC_C Library - source code	$175
EC - full screen editor	$ 85
BASIC_C & EC	$225
BASIC_C & DeSmet C Compiler	$275
BASIC_C & Computer Innovations C compiler	$475
BASIC_C & Lattice C compiler	$550
BASIC_C, DeSmet C Compiler, & EC	$325
BASIC_C, Computer Innovations C compiler, & EC	$525
BASIC_C, Lattice C compiler, & EC	$600

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
/* music.c - plays Looney Tunes versions of Merrily We Roll Along
                and The Merry-Go-Round Broke Down.
                music.c must be linked to sound.o
                A> bind music sound

                Written by Robert T. McCrory
                DeSmet C compiler version
*/


#include "stdio.h"

#define NOTEEND '/'     /* end of note delimiter */
#define NULL '\0'       /* end of string */
#define UPOCTAVE '>'    /* permanently move up an octave */
#define DOWNOCTAVE '<'  /* permanently move down an octave */
#define SHARP '+'       /* make a note a sharp... */
#define FLAT '-'        /* ...or a flat */
#define DOT '.'         /* increase value of note by half */
#define REST 'R'        /* rest - use with note value for all rests */

main()
{
merrily();
merry();
}

/**** Tune 1 ****/
merrily()
{
        playtune("e4./d8/c4/d4/e4/d+4/e4/c4/d4/d4/d4/d1",90);
        playtune("d4./c8/b4/c4/d4/c+4/d4/b4/c4/c4/c4/c1/<g4./g+8/",90);
        playtune(">a4./a-8/<g4./g+8/>a4/a-4/<g4/>d4/d4/d2./<g4./g+8/",90);
        playtune(">a4./a-8/<g4./g+8/>a4/a-4/<g4/>e4/e4/e2.",90);
        playtune("e4./d8/c4/d4/e4/d+4/e4/c4/d4/d4/d4/d2/c4/<g+4/>a4",90);
        playtune("d2/e2/g1",90);
}

/**** Tune 2 ****/
merry()
{
        playtune("g8/g8/>c8/c8/b4/a4/<g4/e2./g4/>c4/b4/a4/a+4/b2.",80);
        playtune("b4/b4/<g4/>a4/<g4/>b4/<g4/>a4/<g4/g8/g8/g4/>a4/b4/c8.",80);
        playtune("d8./e4",80);
}


/**** functions ****/

playtune(tunestring,mm)         /* this plays notes in a string by
                                   splitting the string up between NOTEEND's
                                   and passing the substrings to
                                   playnote for decoding and play */

char *tunestring;               /* note string to be played */
int mm;                         /* metronome in whole notes per minute */
{
        char tempstring[10];    /* holds substrings for passage to playnote */
        int i,j,length;

        length = strlen(tunestring);        /* length of tune string */

        for(i = 0, j = 0; i < length; i++) {
                        /* assign chars to tempstring until note delimiter */
                if((*(tempstring + j++) = *(tunestring + i)) != NOTEEND)
                ;
                else {
                          /* find delimiter, end tempstring and play note */
                        *(tempstring + --j) = NULL;
                        playnote(tempstring,mm);
                        j = 0;
                }
        }
        if(j != 0) {            /* last note if not ended with a / */
                *(tempstring + j) = NULL;
                playnote(tempstring,mm);
        }
}


playnote(notestring,mm)         /* plays a single note based on note and
                duration information in notestring */

char *notestring;               /* single note string to be played */
int mm;                         /* metronome in whole notes per minute */
{
                                /* use UPOCTAVE and DOWNOCTAVE for following notes */
        static float octavefactor = 1.0;

                                /* double index to allow for sharps and flats */
        static int noteindex[] = {
                0,2,3,5,7,8,10
                };

        static int notefreq[] = {           /* base frequency table */
                440,466,494,523,554,587,        /* begins with A-440 */
                622,659,698,740,784,831
                };

        int length,place,durplace,test,freq,duration;
        int index;
        int indexadj = 0;   /* adjustment for sharps and flats */

        char durstring[5];  /* accumulate all numbers for duration of note */
        char toupper(),charatplace,noteletter;
```

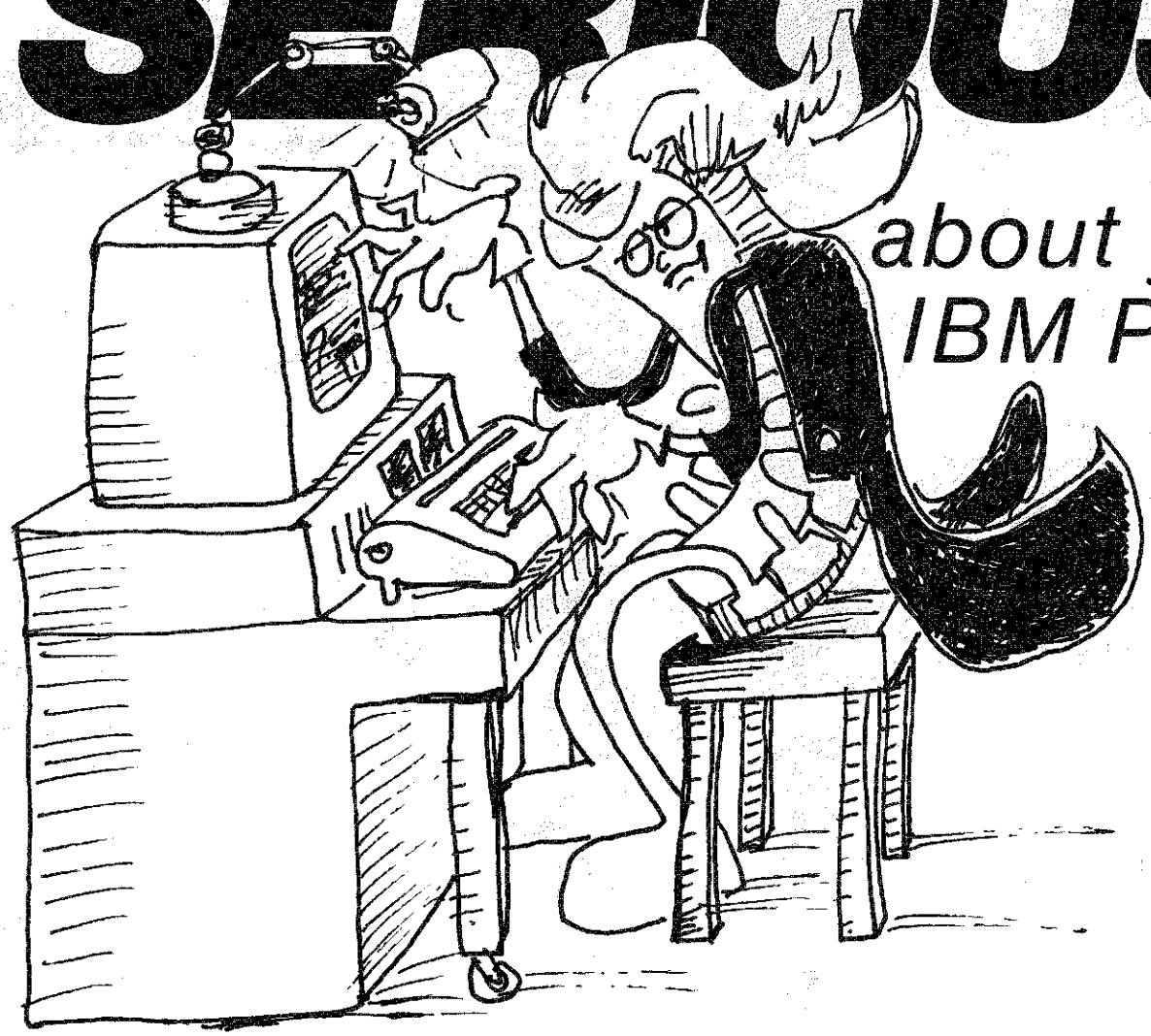
PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
float noteoctavefactor = 1.0;      /* octave factor = SHARP or FLAT */
float dottednote = 1.0;            /* DOT signifies a dotted note */

length = strlen(notestring);       /* length of notestring */

for(place = 0, durplace = 0; place < length; place++) {

                    /* read character in notestring */
        charatplace = toupper(*(notestring + place));

                    /* if alpha, determine index into freq table */
        if(test = isalpha(charatplace)) {
                index = *(noteindex + (charatplace - 'A'));
                noteletter = charatplace;
        }

        else
            /* if not alpha, treat as control or duration character */
                switch(charatplace) {
                        case UPOCTAVE:
                                octavefactor *= 2.;
                                break;
                        case DOWNOCTAVE:
                                octavefactor /= 2.;
                                break;
                        case SHARP:
                                indexadj++;
                                break;
                        case FLAT:
                                indexadj--;
                                break;
                        case DOT:
                                dottednote = 1.5;
                                break;
                        default:
                /* if a digit, accumulate in durstring for duration */
                          if(test = isdigit(charatplace))
                             *(durstring + durplace++) = charatplace;
                                break;

                }
}

        /* decode string. 1 is whole note, 4 is quarter note, etc. */
*(durstring + durplace) = NULL;
duration = atoi(durstring);
duration = (int)(6000.*dottednote/(mm*duration));


if(noteletter == REST)             /* decode pitch */
        freq = 32000;              /* silence at rest */
else {
        index += indexadj;
        while(index < 0) {     /* adjust octaves for sharps - flats */
                index += 12;
                noteoctavefactor /= 2.;
        }
                while(index >= 12) {
                        index -= 12;
                        noteoctavefactor *= 2.;
                }
                freq = (*(notefreq + index))*octavefactor*noteoctavefactor;
        }
        tone(freq,duration);               /* play the note */
}
```

```
;       sound.a  assembly language sound routines for c88
;
;       TONE     produce a tone of specified frequency and duration
;                Usage:  tone(freq,duration) where
;                        freq is frequency of tone generated in Hz
;                        duration is length of tone in hundredths of a sec

        cseg
        public tone_

tone_:  push bp
        mov bp,sp

        mov di,[bp+4]              ; move freq to di
        mov bx,[bp+6]              ; move duration to bx

        mov al,0B6H                ; write to timer mode register
        out 43H,al

        mov dx,14H                 ; timer divisor =
        mov ax,4F38H               ;    1331000/Frequency
        div di
        out 42H,al                 ; write timer 2 count low byte
        mov al,ah
        out 42H,al                 ; write timer 2 count high byte
        in al,61H                  ; get current port B setting
        mov ah,al                  ;    and save it in ah
        or al,3                    ; turn speaker on
        out 61H,al
stall:  mov cx,2801                ; wait 10 milliseconds
spkr_on: loop spkr_on
        dec bx                     ; speaker-on count expired?
        jnz stall                  ; if not, keep speaker on
        mov al,ah                  ; otherwise, recover value of port
        out 61H,al

        pop bp                     ; That's all, folks!
        ret

        end of asm code
        □
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

SERIOUS



about your IBM PC?

***READ** the Definitive Journal for the IBM Personal Computer User.* Too serious about your IBM PC to plow through 3 lbs. of ads...industry history...revealing personal interviews? Consider Personal Computer Age. Written and published monthly by serious computerists for serious computerists, Personal Computer Age presents PC technology from fundamentals to hands-on tweaking...And explains it so you can use it.

Start your subscription now. Personal Computer Age... one year 12 informative issues...just $24.00.

The Definitive Journal for the IBM Personal Computer User
Personal Computer Age

Enter my subscription today!

☐ New
☐ Renewal
☐ Payment enclosed
☐ Bill me

charge my:
☐ Visa ☐ MASTERCARD Exp.Date_____

Card # _____

_____
(authorization signature)

☐ 1 yr. (12 issues) $24.00
☐ 2 yrs. (24 issues) $45.00
☐ 3 yrs. (36 issues) $64.00
☐ Send information on back issues, too!

1 yr subscription. Canada & Mexico $29.00
Other countries via Air . $90.00
Surface mail $40.00
Payment made in U.S. funds drawn on U.S. bank.

Name _____

Address _____

City _____ State _____ Zip _____

Department 6083
P.O. Box 70725
Pasadena,
California 91107
(800) 824-7888
operator 77

The Definitive Journal for the IBM Personal Computer User
Personal Computer Age

Department 6083
P.O. Box 70725
Pasadena,
California 91107
(800) 824-7888
operator 77

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

A PRINTER CONTROL DATABASE

by Jim Bohle

Routing reports and other information to a printer is relatively simple using dBASE II. Probably the simplest method for printing dBASE II information is to issue the "set print on" command and follow this with a series of "display" or "?" commands, such as:

```
set print on
? " Client's List"
? NAME, ADDRESS, PHONE
```

Printing stops whenever the "set print off" command is issued. If you want a report to go to the printer without appearing on the screen, you can give the command "set console off" just prior to "set print on." Remember to issue a "set console on" when you want the screen to be active again.

If you want greater control of print formats and alignments, then the combination of "set format to print" and "@ row,col say..." can be used. The "set format to print" command works similarly to the "set print on," except that "set format" sends "@ row,col say..." commands to the printer as well. The above example could have been printed with more specific format instructions:

```
set format to print @ 01,30 say "Client's List"
@ 05,05 say NAME
@ 05,20 say ADDRESS
@ 05,40 say PHONE
set format to screen
```

Without "set format to print" this information would normally appear on the console screen at the locations designated by "row,col." A "set format to screen" command redirects the information back to the console.

Information from dBASE II can be printed indirectly by means of a print spooler (e.g., MSDOS PRINT command) or a word-processing system if data is first routed to a disk file. This is a useful technique if the report is quite long,requires further editing, or will be saved for future use. Issuing the dBASE II commands

```
set alternate to REPORT.PRT
set alternate on
. . . . . .print the report
set alternate off
```

will direct all information normally written to the screen or printer to the specified disk file, REPORT.PRT. The "set alternate off" command closes the disk file and redirects the information back to the console screen.

Besides the printing of reports, you might want to use some of your printer's advanced features for emphasis in other applications. Some reports require special formating or presentations, such as 12-pitch and double-width characters, underlining, correspondence-quality print, special line spacing, or graphics control. Most of the popular printers, such as the Okidata, Epson, or NEC series, provide an extensive set of special printing features. These enhancements are usually invoked by routing specific instructions to the printer during the printing process. The instructions are essentially sequences of one or more control characters which the printer is designed to recognize. Usually the printer manual describes in full detail the specific control code values and sequences used to invoke these special functions. The Okidata ML92 manual, for example, gives the ASCII character, decimal and hexadecimal equivalent values for each control sequence. You can instruct the ML92 to print characters in 12-pitch (12 characters per inch) by sending the printer the single ASCII special control character "(FS)" or its equivalent character decimal value 28. Since most of the special characters are not ordinarily generated from a single key on the keyboard, the dBASE II "chr()" function can be used to generate the character value. The following sequence of dBASE II commands will print the message below in 12-pitch characters on an Okidata ML92 printer:

```
set print on
? chr(28)
? "These are printed at 12 characters per inch"
```

Issuing the command "? chr(30)" will return the printer to the normal 10-pitch mode.

The "?" command presents two special problems when you attempt to send a sequence of control characters to the printer. First, since "?" generates a carriage return and line feed prior to printing, the paper advances a line. This is a problem if you're not expecting an extra line feed in your report. More important,if you're changing modes in the middle of a line, such as with underlining or subscripting,.you certainly do not want the underlined word to appear on the next line. Fortunately, dBASE II provides a way to overcome this with "??", which suppresses the carriage return and line feed. Therefore, the second line in the sequence above should be changed to:

```
?? chr(28)
```

The second problem appears when you attempt to use control sequences involving more than one character. For example, the Okidata ML92 provides a correspondence-quality print mode which is invoked by the sequence "ESC 1"

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

or "chr(27), chr(49)." Initially, it seems either of the commands,

?? chr(27), "1" or ?? chr(27), chr(49)

should work. However, the display command inserts a character space between each item, and the printer actually receives the three character sequence "ESC SPACE 1," which it does not recognize. Two solutions are possible: use "set raw on" to suppress the unwanted characters, or concatenate them into a single string, which eliminates the intruding space:

```
set raw on
?? chr(27), "1"
or
?? chr(27) + chr(49)
```

Since the printer control sequences are really strings, they can be created by concatenating them and conveniently storing them to memory variables such as:

```
store chr(28) to MPC:CPI12
store chr(27) + chr(49) to MPC:CORRES
```

If we wanted to print a report using the Okidata's correspondence-quality mode, the above memory variables would work very nicely (12 pitch is included, too.):

```
set print on
?? MPC:CORRES + MPC:CPI12
? "Client's List"
? NAME, ADDRESS, PHONE
```

A set of memory variables containing the printer control definitions could be saved to a memory file and then recalled for later use with the following:

```
save to PRINTERD all like MPC:*
..... restore from PRINTERD
```

The next logical step for printer control enhancements is to create a database containing the control sequences. By employing a database, the control codes can be readily available for use in any command file. Furthermore, printer functions can be grouped into sets for special uses (e.g., a standard set for reports, a special set for graphics applications). If you have more than one printer, specific printer control definition memory files can be created for each printer since the control sequences are probably different. Programs can prompt for the specific printer control file to be used.

A printer control definition database should include for each special printer function a description, a reference name for the control sequence and the actual control code values. Such a database could be defined as:

```
01 PCC:NAME C 06 reference memory variable name
02 PCC:DESC C 20 printer function description
03 PCC:VAL1 N 03 control code value 1 (eg. 27)
04 PCC:VAL2 N03 control code value 2 (eg. 49)
05 PCC:VAL3 N 03 control code value 3 (eg. 255)
06 PCC:VAL4 N 03 control code value 4
07 PCC:VAL5 N 03 control code value 5
08 PCC:VAL6 N 03 control code value 6
09 PCC:VAL7 N 03 control code value 7
```

The function description, PCC:DESC, can be approximately 20 characters long. The reference name, PCC:NAME, is a six-character string which associates a memory variable with a printer function. The control code values, PCC:VALn, are three digit numeric fields which facilitate entry of the decimal values. A few of the more useful, nongraphics functions and control sequences for the Okidata ML92 printer are given on the following page. The control definitions can be entered very quickly by simple APPEND and EDIT commands once the database has been created.

The values PCC:VAL5 through PCC:VAL7 are generally used with more advanced graphics definitions. The value 255 is used as a delimiting character indicating the end of a control value.

Once the database is complete, it is easy to convert the definitions into a memory variable file which can be accessed from a dBASE II program. A small system designed to support the printer control functions might look like this:

```
A — Add Definitions
C — Change Definitions
D — Delete Definitions
L — List Definitions
B — Build Definitions File
P — Printer Test Procedure
E — End Session
```

The file maintenance functions — Add, Change, Delete, List — are relatively straightforward, so they will not be discussed here. The PCDMEM routine, which follows, converts the printer codes into a memory variable file. The codes can then be read in one at a time and put together into a string. Since control sequences are variable in length, a loop is employed to concatenate values until the delimiter (255) is encountered or until all seven values have been used. Temporary fields are used to make the transfer (M:NAME and M:NUM).

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
store ' ' to M:REF
store 'PCC:VAL' to M:NAME
store 0 to M:NUM

do while CONTINUE .and. M:NUM<8
   store 1 + M:NUM to M:NUM
   store M:NAME + str(M:NUM,1) to M:REF

   if &M:REF<>255
      store M:STRING + chr(&M:REF) to M:STRING
   else
      store F to CONTINUE
   endif
enddo
```

The reference to "&M:REF", using the macro substitution function "&", yields the field name of one of the control code values (eg., PCC:VAL1). For example, if M:NAME contains the string 'PCC:VAL' and M:NUM contains the numeric value "1", they can be put together to form PCC:VAL1.

Once a control sequence string has been formed, it is stored to a memory variable, PCC:NAME. A prefix of "MPC:" (for memory printer control) is concatenated to each reference name for convenience and consistency. The control sequence string (less the first character, which is a space) is then stored to the memory variable, @M:MEMVAR:

store M:PREF + PCC:NAME to M:MEMVAR
store $(M:STRING,2,10) to &M:MEMVAR

Finally, a complete set of codes is stored to a file for subsequent use by other programs:

save to &M:MEMFILE all like &PREF + "*"

The test routine, PCDTST, included below with sample printing, demonstrates the use of some of the special printer

NONGRAPHICS FUNCTIONS AND CONTROL SEQUENCES FOR THE OKIDATA ML92 PRINTER

PCC:NAME	PCC:DESC	VAL1	VAL2	VAL3	VAL4
LF	Line Feed	10	255		
CR	Carriage Return	13	255		
FF	Form Feed	12	255		
RELSUP	Release Suppress	17	255		
SUPRSS	Set Suppress Mode	19	255		
CPI10	10 Chars Per Inch	30	255		
CPI12	12 Chars Per Inch	28	255		
CPI17	17 Chars Per Inch	29	255		
DWIDTH	Double Width Chars	31	255		
CLEAR	Clear Buffer - Reset	24	255		
LFNOCR	Line Feed - No CR	27	18	255	
MATRIX	Dot Matrix Printing	27	48	255	
CORRES	Correspondence Print	27	49	255	
SETTOF	Set Top of Form	27	53	255	
LPI6	6 Lines Per Inch	27	54	255	
LPI8	8 Lines Per Inch	27	56	255	
ULBEG	Begin Underlining	27	67	255	
ULEND	Stop Underlining	27	68	255	
LPPAGE	Set Lines Per Page	27	70	255	*
ENHNCE	Print Enhanced	27	72	255	
EMPHZE	Print Emphasized	27	84	255	
STOPEE	Stop Enhnce / Emphze	27	73	255	
SUPERS	Print Superscripts	27	74	255	
SUPEND	Stop Superscripts	27	75	255	
SUBSCR	Print Subscripts	27	76	255	
SUBEND	Stop Subscripts	27	77	255	
CSPACE	Set Char Spacing	27	78	255	**
LSPACE	Set Line Spacing	27	37	255	**
LSKIP	Skip N Lines	27	11	255	*

* add 00 to 99 lines as "nn" to control definition
** add 1 to 127 space increments to control definition

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

functions. Note that the set of variables is retrieved by means of the "restore" command. The routine also demonstrates the optional use of an alternate file for output. Some of the control sequences require that additional information be supplied. The function to set line spacing, LSPACE, requires that a value between 1 and 127 be sent to the printer to indicate the width. Since this is defined in the printer manual as a binary number, a character representation for the number is concatenated to the end of the control sequence using the "chr()" function, such as

?? MPC:LSPACE + chr(36)

for a value of 36.

Printer control definitions and report text can be mixed in a single line, as illustrated in the case of underlining and subscripting. As mentioned previously, it is important to use the "??" command to suppress the carriage return and line feed characters. Multiple definitions can also be combined, as in the case of correspondence-quality print and special line spacing. Using MPC:CLEAR to reset the printer merits special attention, since it totally clears the print buffer. Because the "?" command issues a carriage return line feed prior to printing the text, an extra "?" must be included in a program in order to print out the last line.

The printer control database and memory file have been used with several dBASE II applications with great success. Since the definitions reside in a file, they are very flexible. Many different sets can be created depending on your application needs and the number of printers you have.

```
******  PCDMEM.PRG  --  Build Printer Control Definitions Memory File   ******
*                                                                              *
*       The PCDMEM.PRG routine creates a dBASE memory (.MEM) file using        *
*       the current set of control code definitions found in the speci-        *
*       fied printer database.  The OKIDATA database described below           *
*       contains a portion of the code sequences use to control the            *
*       OKIDATA Microline 92 and 93 printers.                                  *
*                                                                              *
*                              OKIDATA.DBF                                     *
*           ------------------------------------------------------             *
*           01   PCC:NAME  C  06   Reference Name, Memory Variable             *
*           02   PCC:DESC  C  20   Printer Control Def Description             *
*           03   PCC:VAL1  N  03   Control Code Value 1  (e.g. 27)             *
*           04   PCC:VAL2  N  03   Control Code Value 2  (e.g. 49)             *
*           05   PCC:VAL3  N  03   Control Code Value 3  (e.g.255)             *
*           06   PCC:VAL4  N  03   Control Code Value 4                        *
*           07   PCC:VAL5  N  03   Control Code Value 5                        *
*           08   PCC:VAL6  N  03   Control Code Value 6                        *
*           09   PCC:VAL7  N  03   Control Code Value 7                        *
*                                                                              *
*       The routine begins by prompting the user for the names of the          *
*       printer control definitions database (M:DBFILE) and resulting          *
*       memory file (M:MEMFILE).  Printer control definitions are read         *
*       sequentially (from first to last).  For each definition record,        *
*       the control codes (PCC:VAL1 to PCC:VAL7) are combined to form          *
*       a single string.  The control codes are defined as type 'N' so         *
*       they may be entered by their decimal character value (e.g. 12          *
*       for FF or Form Feed).  The control string is formed by concat-         *
*       enating the character representation (via the CHR function) of         *
*       each control code value.  Resulting control strings have a             *
*       length of 1 to 7 characters.  The value 255 is used to delimit         *
*       the end of control sequences with less than 7 code values. The         *
*       example given above (27, 49) is used to place the OKIDATA ML           *
*       92 and 93 printers in correspondence print mode.  The control          *
*       strings are 'stored to' a memory variable referenced by the            *
*       name contained in PCC:NAME.  Therefore, the user is free to            *
*       associate any name with a given printer control definition.            *
*       The entire set of printer control strings is 'saved to' the            *
*       designated memory file for use in other dBASE command files.           *
*                                                                              *
********************************************************************************
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
*      perform initial housekeeping and set options

erase
clear
set talk off
set carry off
set colon off
set bell off
*      initialize memory variables

store '           ' to M:DBFILE, M:MEMFILE
store '           ' to M:STRING, M:REF, M:MEMVAR
store 'PCC:VAL' to M:NAME
store 'MPC:'    to M:PREF
store 0 to M:NUM
store 'N' to VERIFY
store ' ' to ANY:KEY

*      display standard PCDEFS screen; select database and
*      memory file name; loop until user verifies names

do PCDEFS.SCR

do while (VERIFY='N')
    @ 05,18 say "Build Printer Control Definitions Memory File"
    @ 08,23 say "Printer Database Name    "   get M:DBFILE  picture 'XXXXXXXXXX'
    @ 10,23 say "Definitions Memory File "   get M:MEMFILE picture 'XXXXXXXXXX'
    @ 12,27 say "Verify (Yes or No)       "   get VERIFY    picture '!'
    read

    if .not. file(M:DBFILE)
         @ 23,05 say trim(M:DBFILE) + " database not found!"
         @ 23,50 say "Press any key to continue "      get ANY:KEY picture "X"
         read
         @ 23,00
         store 'N' to VERIFY
    endif
enddo

use &M:DBFILE
goto top

*            -------  main routine  -------
*      read each printer control definition in database

@ 16,13 say "processing .... printer control definitions read --> "
do while .not. eof

    store T   to CONTINUE
    store ' ' to M:STRING
    store 0   to M:NUM

*      build control code string by concatenating values
*      stored in PCC:VAL1 to PCC:VAL7.  Access to each field is
*      made by forming variable name from M:NAME and M:NUM

    do while CONTINUE .and. M:NUM<8
         store 1 + M:NUM to M:NUM
         store M:NAME + str(M:NUM,1) to M:REF
*      check for delimiter character specifying end of control codes
*      if not delimiter, add new character code to end of string

         if &M:REF<>255
              store M:STRING + chr(&M:REF) to M:STRING
         else
              store F to CONTINUE
         endif
    enddo

*      store control code string to memory variable name
*      contained in database field PCC:NAME with prefix
*      'MPC:', skip blank character at beginning of M:STRING

    store M:PREF + PCC:NAME to M:MEMVAR
    store $(M:STRING,2,10) to &M:MEMVAR

*      read next printer control code definition
    @ 16,66 say # using '##'
    skip

enddo

*      save all printer control codes as stored in the designated
*      memory variable to the specified .MEM file for subsequent use

@ 18,15 say "saving definitions to " + trim(M:MEMFILE) + ".MEM .... "

save to &M:MEMFILE

@ 18,56 say "completed!"

*      perform final housekeeping before exit
use
release all
@ 23,00

return
```

```
***  PCDTST.PRG  --  Printer Test Using Control Definitions Memory File ******
*                                                                            *
*       The PCDTST.PRG routine tests the sample Printer Control Def-         *
*       initions for the OKIDATA ML 92/93 printers as defined in the         *
*       OKIDATA.MEM file and derived from the OKIDATA definitions            *
*       database.  Approximately half of the printer control functions       *
*       are demonstrated.  The routine begins by retrieving the printer      *
*       control memory variables (MPC:xxxxxx) by means of the command        *
*       'restore from OKIDATA.MEM'.  The option to send the test print       *
*       directly to the printer or to a disk file for subsequent print-      *
*       is presented and the routine issues a 'set print on' or 'set         *
*       alternate on' accordingly.  At this point the user is ready to       *
*       proceed with printing.  Please see sample print lines below.         *
*                                                                            *
******************************************************************************
*      perform initial housekeeping and set options

erase
clear
set talk off
set carry off
set colon off
set bell off
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
*      initialize memory variables

store 'N' to VERIFY
store 'P' to M:OUTP

*      display standard printer control definitions screen
*      prompt for destination of print test

do PCDEFS.SCR

do while (VERIFY='N')
   @ 05,30 say "Print Test Procedure"
   @ 08,25 say "Report to (P)rinter or (D)isk? "  get M:OUTP picture '!'
   @ 10,25 say "Verify  (Y)es or (N)o          "  get VERIFY   picture '!'
   read
   if (M:OUTP<>"P" .and. M:OUTP<>"D")
      store "N" to VERIFY
   endif
enddo

*      set up printer or disk file for printer test report

if M:OUTP='P'
   set print on
else
   set alternate to PCDTST.PRT
   set alternate on
endif

*      retrieve the OKIDATA printer control definition memory variables

restore from OKIDATA.MEM



*      begin test procedure; set printer default options (CLEAR)

@ 16,16 say "begin printer control definitions test procedure"
set console off

?? MPC:CLEAR

*      test various character width / pitch options

?  "This text is printed with a pitch of 10 characters per inch (default)"
?? MPC:CPI12
?  "This text is printed with a pitch of 12 characters per inch"
?? MPC:CPI17
?  "This text is printed with a pitch of 17 characters per inch"
?? MPC:CPI10
?  "Return to standard pitch of 10 characters per inch"

*      test double with character options

?? MPC:CPI10 + MPC:DWIDTH
?  "This is DOUBLE width 5 cpi"
?? MPC:CPI12 + MPC:DWIDTH
?  "This is DOUBLE width 6 cpi"
?? MPC:CPI17 + MPC:DWIDTH
?  "This is DOUBLE width 8.5 cpi"
?

*      demonstrate variety of special printing styles

?? MPC:CLEAR
?  "This text appears in the standard dot matrix, data processing format"
?? MPC:CORRES
?  "This text is printed in the correspondence quality print format - sharp"
?? MPC:MATRIX
?  "This text is again printed in the dot matrix format"
?? MPC:ENHNCE
?  "This is a sample of HALF-DOT enhanced printing"
?? MPC:EMPHZE
?  "This is a sample of HALF-DOT emphasized printing"
?

?? MPC:CLEAR
?? MPC:CORRES
?  "Here are " + MPC:ULBEG + "underlined characters" + MPC:ULEND + " too!"
?
?  "Here are " + MPC:SUBSCR + "subscripts" + MPC:SUBEND + " and more"
?
?  "Here are " + MPC:SUPERS + "superscripts" + MPC:SUPEND + " and less"
?

*      special print control functions requiring parameter values

?? MPC:CLEAR


?  "The next line will be printed after skipping three lines"
?? MPC:LSKIP + "03"
?  "This line is printed after skipping three lines from the previous line"
?  "The next three lines will be printed with line spacing of 36/144 inch"
?? MPC:LSPACE + chr(36)
?  "This line is 36/144 or 1/4 inch below the last one"
?  "So is this one.  This spacing is equivalent to 4 lines per inch"
?  "Normally lines are printed 6 per inch with a standard option of 8 LPI"
?? MPC:LPI8 + MPC:CORRES
?  "See -- the next two lines are printed with a spacing of 8 lines per inch"
?  "in the correspondence print mode.  Pretty compact!!"
?
?? MPC:CLEAR
?  "Back to standard mode -- end of printer control definitions test"
?

*      perform final housekeeping -- turn off printer/close disk file

set print off
set alternate off

set console on
@ 18,35 say "Complete!"
release all
@ 23,00
return
```

□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

TOOLS, TOOLS2, AND VIEW MANAGER

review by Chet Floyd

My interest in these products began several months ago. I had begun coding a set of tools from Software Tools in Pascal by Kernighan and Plauger. This is an excellent book, but I found a couple of obstacles: the K&P Tools are written in ISO Pascal and so do not take advantage of the IBM PC, and the programs I developed seemed to run slower than expected. Fortunately I had not done a whole lot of coding before Blaise Computing's TOOLS came to my attention.

Blaise Computing has developed a set of PC-DOS/MS-DOS productivity tools that every Pascal or C programmer should be aware of. You should find, as I have, that use of any one of the packages results in reduced development cost, added capabilities for the application, or both.

It is worth noting at the outset that the Blaise Computing products are designed specifically as productivity aids to fill the gaps between Pascal and the capabilities of MS-DOS and the IBM PC. They are not the "happy accidents" of other projects. This is apparent in the user documentation, the commented source code, the error checking, and the way in which the wide variety of functions and procedures work so well together. It also shows in the support: I have received prompt, effective answers to my questions.

Initially written for IBM Pascal, the three packages TOOLS, TOOLS2, and VIEW MANAGER have just been rewritten for Microsoft (Lattice) C. In addition, programmers who have migrated from IBM to Microsoft Pascal will be pleased to find that Blaise Computing now is shipping object files that are compatible with the Microsoft Pascal library. Previously, batch files were included to recompile the source for MS-Pascal, making the transition easier. Presently, these are the only compilers supported.

Just what is this set of aids? Briefly, Blaise Computing offers three levels of productivity tools, each one complete in itself but building upon the capabilities of the previous ones. This means you can purchase them incrementally to match your needs.

TOOLS provides 51 functions and procedures that are so useful I consider them extensions to Pascal. This set of routines is comprised of six groups. One group provides the screen management and access so sorely missing in IBM Pascal if you wish to exploit the PC's capabilities. Another expands string handling capabilities by providing translation, conversion, and trimming.

For example: have you ever wondered why Pascal never provided the advanced string handling of BASIC, like MID$, INSTR, etc.? The question also occurred to Blaise Computing. This idea of functionality is part and parcel of TOOLS. Their "cvtzz" routine in TOOLS can be used to trim tabs or blanks from the left or right side of a string, convert it to upper or lower case, and remove special characters, all with one call.

And would you like to display date and time as "Friday February 3, 1984 4:22 PM"? It's almost automatic with TOOLS.

If you have needed to parse command line parameters with Pascal, you know how laborious that can be. From Software Tools in Pascal I coded up something that kind of worked in IBM Pascal. When I discovered "comlinxx" and "cparsexx" in TOOLS, I laid aside my precious effort and never looked back.

There are four more groups of routines that do graphics, message display, prompting, and math, all with equal facility. If you are doing any programming at all in Pascal or C, do look seriously at TOOLS.

TOOLS2 exploits DOS 2.0 with a suite of 32 routines that traditionally have been implemented in assembler. Yet with just one machine language module, called a DOS Gate, each facility is provided in Pascal, and they execute quickly. You get memory and program management for program chaining and data storage outside the 64K data space the compiler allows; file handling to take advantage of 2.0 buffering and paths; directory maintenance to let you do things like set files to read only, or even manage and manipulate sub-directories from within your Pascal program. There is also a set of routines that by all rights should be part ofany language designed for development of commercial programs: setting and returning the system date and time, and checking available disk space.

A sample program from TOOLS2 — one of eight in the suite of three packages — demonstrates file handling speed with a program in the classic read(ch) — write(ch) form. This is a compression filter that runs like the wind. It outperforms the K&P example by a huge factor. The secret of Blaise Computing's blazing speed is that the disk is directly read into and written from your buffers, and you have control of the buffer size.

If you read Dr. Dobbs Journal, you may have seen in a recent issue an article describing how to invoke one program from another. If you had TOOLS2, you would just use the "shrinkmm" and "execmm" procedures.

And you wouldn't have to worry if there wasn't enough memory to invoke a program, Blaise Computing code has

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

extensive error checking, supplied as a byte or word return value from a procedure. Blaise Computing engineers obviously have a clear insight as to when an error code should be available: when the documentation suggests checking the error code, please do. This advice has saved me hours of potential debugging.

Here is a brief example that uses TOOLS and TOOLS2 to allocate a 10K data segment, invoke another program, and report any errors to the terminal. The (n)s are an indication for TOOLS (1) and TOOLS2 (2).

```
(2)shrinkmm(errorw);                    ! free unneeded mem
   if (errorw <>0) then begin
(1)          cursetvv (21,2);
(1)          attribvv (15,0,' ' 30);    ! hi intensity
             write ('Memory Shrink error =',errorw:3);
(1)          pauseuu (NULL);
             end;

(2)allocmm (640, blkseg, blksize, errorw);
   ! Ask for 16*640; segment address returned in blkseg
   if (errorw <>0) then begin
             cursetvv (21,2);
             attribvv (15,0,' ' 30);    ! hi intensity
             write ('Memory Alloc. error =',errorw:3);
             pauseuu (NULL);
             end;

(2)execmm (prog_name, cmd_line, errorw); ! Away we go...
   if (errorw <>0) then begin
             cursetvv (21,2);
             attribvv (15,0,' ' 30);    ! hi intensity
      write ('Error',errorw:3,' invoking ',prog_name);
         pauseuu (NULL);
         end;
```

The source code for TOOLS and TOOLS2 is included in the purchase price. Since the object code is also included, you don't need to compile unless you are using the Microsoft Pascal compiler. With the growing popularity of MS-Pascal, however, Blaise Computing has begun shipping object modules that will link with MS-Pascal. With very minor exceptions, I have not had any need to alter the Blaise Computing code. It has been called "carefully crafted," and it is a delightto read and work with.

VIEW MANAGER is a comprehensive screen development facility. You get a screen painter, utilities to manage the screen database you build, and routines to access and manipulate the screens from your program.

The starting point, whether you are creating or modifying a screen, is the Screen Painter. This utility is easy to use — simply "paint" fields and headings where you want them, in their attributes. Effective use of the function keys controls attribute selection, and a dozen or so command keys provide editing as well as box drawing. When done, the screen is "compiled" into individual fields and saved to the screen database.

Next, you will want to run the utility that edits field attributes of the screen. This is where data fields are distinguished from descriptive data, and where first-level editing is specified for each data entry field. First-level editing includes upper or lower case, alpha or numeric, left or right justification, zero fill, field duplication, and required or optional field. These edits may be combined. With this utility you can alter the names of the screen fields that VIEW MANAGER has assigned as a function of the fields' coordinates. I remove all field names and replace the data entry fields with meaningful symbols.

But wait, there's more! Another utility builds and prints documentation of your screens. Additional utilities copy screens to new databases and reorganize existing databases.

The screen database is indexed for fast access, and Blaise Computing has done it right by providing proper utilities to manage the files that result. These utilities and their ease of use allow you to be productive without investing in hours of familiarization. They are sufficiently clear to be used by an assistant, if you enjoy such a luxury, without close supervision. I showed VIEW MANAGER to a colleague who had just started to work with another major C utility library. He turned green with envy. Good news, Colin: by the time you read this, the Blaise Computing packages should be available for Lattice C.

The productivity of the VIEW MANAGER utilities is impressively demonstrated by the fact that the utilities were created using the VIEW MANAGER. They make an impressive demo of the VIEW MANAGER capabilities.

A major VIEW MANAGER capability we haven't touched on yet is the Help facility. Each screen you make may have an optional Help file associated with it. These files are ASCII, and may be built with most any editor. VIEW MANAGER handles the task of pagination and prompting when they are invoked.

Fourteen procedures are provided for using VIEW MANAGER in your programs. The obvious housekeepers like open, close, display, and read screen are there. Also included are procedures to page in up to three screens, for lightning-fast screen changes. Other procs invoke Help and give access to individual field parameters. This latter is most useful for dynamically creating subsets of the screen. It is possible, for example, to alter individual field attributes so that subsequent edits will allow your user access to only the fields that still need attention, even though all fields remain displayed on the screen.

Communicating data between your program and a VIEW MANAGER screen is possible in both directions and is buffered through a character super array. Data entry fields on the screen are concatenated into this array. Consult the screen documentation you created to find the offset of each field in the array. If you dynamically alter the screen to exclude

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

certain fields, these fields just disappear from the buffer, so any fields to the right will have different offsets.

Another characteristic of communicating data with VIEW MANAGER is that VIEW MANAGER stays in control until the operator decides the screen is complete. Blaise Computing calls this 'block mode data entry.' Thus all data entry fields return in the buffer at once. Keep in mind that the first-level editing has taken place, so what remains to be done is to transfer portions of the buffer to your supplementary editing routines or whatever. You tell VIEW MANAGER what keys the operator can use to return from the screen. Besides the Enter key, any of the 40 F-key possibilities can be specified. You can even toggle field verification on and off.

So far we haven't said a word about the editing functions that VIEW MANAGER contains. There are five Field Positioning keys that allow navigation to any field that will accept data. Eight Field Editing keys perform cursor movement and character editing, including field deletion. This is a reasonable but not exhaustive set of editing capabilities. There is, for instance, no way to move to the end of a field with one key stroke. One advantage of a limited set of editing keys, however, is that performance is fast.

If you wish to expand VIEW MANAGER's repertoire, the source code is available at modest extra cost. VIEW MANAGER is as cleanly constructed as the TOOLS, and I have made modifications quickly and with the expected results, the dreaded 'ripple effect' is not in evidence.

A word about the way Blaise Computing implements these programs may be of interest. We all feel comfortable and familiar with program Modules, but the eyes tend to glaze over when the topic of Units, Interfaces and Implementations arises. If you like to learn from examples, you will find that Blaise Computing software serves as excellent source of reference material. Microsoft Pascal provides complete facilities through their Unit compiland concept. Why should you care about Units? Because they offer the only way to obtain typing protection between compilands. And if you don't know why you need this protection, you can just continue to wonder and Debug your parameter lists when they don't live well with each other. (Hint: Linkers don't know Types from Hot Rocks.)

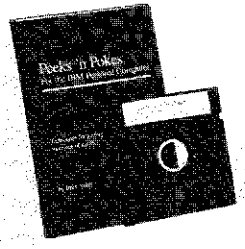
Another factor contributing to the effectiveness of a product is the documentation. Blaise Computing has also done an excellent job here. The documentation is well organized and not only covers the details of each procedure, but discusses the general organization of the product, the sample programs, and linking.
The meat of the documentation is, of course, the detailed procedural descriptions. Each consists of a short discussion of the proc, a prototype statement, detailed description of the formal parameters, error codes returned (if applicable), and either a code fragment for illustration or reference to a sample program. Each description uses as much space as is needed: some are less than half a page, others run to several pages.

The documentation is 8-1/2 by 11, so any changes you make can be easily filed in the manual, and it comes in a three-ring

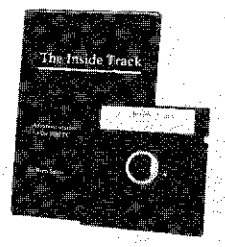
Know Thy PC!

Are you writing programs in BASIC or Pascal? The popular **Peeks 'n Pokes** has a disk with 58 programs and a 38-page manual that helps you get 'underneath the covers' of the PC. Learn how to use PEEK, POKE, INP, OUT, and DOS/BIOS function calls to do what you want, fast! Do you want to perform functions not available from BASIC or Pascal? It's all explained in the manual and demonstrated in the sample programs. Source code included!

Peeks 'n Pokes shows you how to:
- Access the system's configuration
- Unprotect BASIC programs
- Scroll part or all of the screen
- Access the file directory
- Logically swap printers
- Read and change the keyboard
- Find more Peeks and Pokes
- And much more... for only $30.00



Want to know more? **The Inside Track!** is a collection of advanced utilities for the PC programmer. It contains a disk with 61 programs, a 42-page manual, and a fold-out memory map that helps you get better performance from the PC. With this package you can give your programs assembler-assisted speed from high-level languages, get control over memory, customize and control the PC, and more. Some programs require DOS 2.00. Source code included!



The Inside Track! shows you how to:
- Read/write files as fast as DOS
- Display data on the screen faster
- Reserve memory for your use
- Copy memory to another location
- Copy-protect your programs
- Load large programs faster
- Control the keyboard settings
- And much more... for only $45.00

MasterCard and VISA accepted. Shipping charges: $2.50 per order for UPS; $2.50 per item for First Class Mail to USA and Canada; $6.00 per item for Air Mail outside USA and Canada. Dealer inquiries invited.

Data Base Decisions • 14 Bonnie Lane • Atlanta, GA 30328 • 404/256-3860

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

vinyl binder. I have noticed only two minor typos in the three volumes of documentation that comprise the set, and one was a propagation of a muddy description in the IBM Tech Reference manual: the "modevv" function actually returns 7 for a Monochrome monitor.

As another measure of Blaise Computing's attention to detail, a diskette of formatted source is provided in addition to the actual source. Simply dump this diskette to a printer, and you have a paginated reference list of the source.

No product could be so effective without some real warts, right? I do not earn my living by writing reviews, but by developing effective code, so I am pleased to report that warts in these Blaise Computing products, which would be a cost to me, are nonexistent. I did find a couple of obscure and minor bugs in VIEW MANAGER, which are now fixed. To illustrate Blaise Computing's responsiveness, I phoned in two cosmetic bugs and received a reply by phone two working days later. One fix was a trivial code change; the other resulted in a documentation change. These and other fixes are currently being disseminated to registered users, on Blaise Computing's initiative.

There is a wish list of improvements, however, which has come from my daily use of the Blaise Computing code over the last several months:

1. Resolve the inconsistencies in the documentation of error codes. Sometimes they are bytes, sometimes words, and the documentation clearly states which, but they all use the symbol "ercode." I finally just made symbol designations "errorb" and "errorw."

2. Assemble the ASM modules with NAME or TITLE so my librarian can catalogue them. I have had this problem with other vendors' object code also, but at least Blaise supplies the source, so reassembly quickly resolves this problem.

(Really heavy stuff, huh?)

3. Provide a better way to symbolically deal with fields within the VIEW MANAGER screen buffer. I came up with my own solution, learning something in the process, but I had to fool Pascal in the bargain. And we all know it's not nice to fool Pascal.

4. Add the following Field editing capabilities to VIEW MANAGER: cancel the results of field edit — i.e., restore the field, as long as the cursor has not moved out of the field; delete to end of field from current cursor position; skip to end of field; tab within a field.

5. Extend the TOOLS to support communication buffers. At a minimum, the support BASIC offers should be provided. I hear rumblings that Blaise Computing is looking for the way to provide communications support, and do it right.

6. Offer the indexing and linked-list processing routines used in VIEW MANAGER as a separate TOOLS product.

If you write to Blaise Computing, they will send you two meaty pamphlets that detail the specific functions of the packages. There is also a demo package available for VIEW MANAGER. It consists of the actual VIEW MANAGER manual, a demo program, Screen and Help files and the Utility programs with disk write crippled.

What is the cost of using these Blaise Computing products? First, there is the purchase price: $125.00 for TOOLS; $100.00 for TOOLS2; $275.00 for VIEW MANAGER; $425.00 for VIEW MANAGER with source; and $35.00 for the VIEW MANAGER demo. Tell your financial folks that the whole set breaks down to less than seven bucks a routine. If you don't need VIEW MANAGER, figure less than three bucks.

There is no license fee or royalty for code you develop with these products. The usual restrictions apply to the source code, i.e., thou shalt not distribute it to thy clients.

Also, your programs will grow in size with the use of these products. TOOLS and TOOLS2 may take 4K to 15K, depending on how many units get linked; VIEW MANAGER will cost perhaps another 15K, including buffers.

A potential cost may be in diskette handling. The .OBJ files for TOOLS total about 23K, TOOLS2 is about 26K, and VIEW MANAGER is about 18k. If you can catalogue these into a library, you will be ahead of the game. But if you are tight on disk space now, these products, or any others that require disk space for that matter, are not going to help the situation. Since I use an XT, I will beg off the space issue. Perhaps someone out there has developed some disk shuffling techniques that can be shared with the world.

Blaise Computing has produced a quality set of tools for the Pascal, and now C, programmer. They are reasonably priced and efficiently implemented. They are powerful while being unobtrusive and straightforward to use. I believe they accomplish their objective magnificently: to allow us to produce better code with less effort.

Blaise Computing
(415) 540-5441
2034 Blake Street, Berkeley, CA 94704
□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

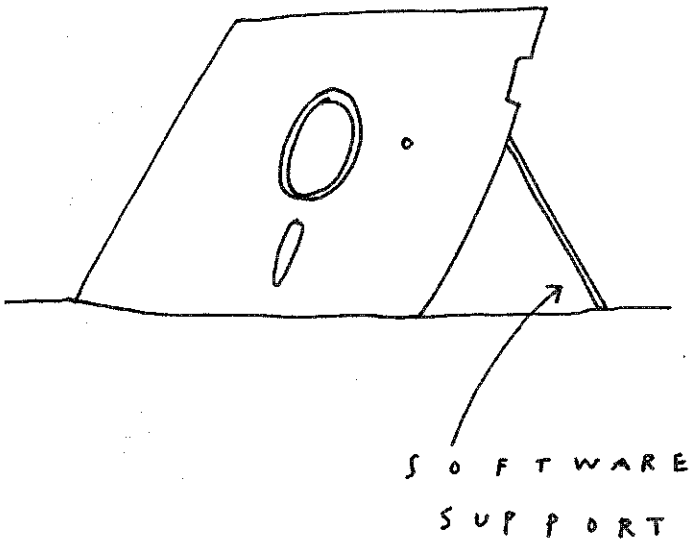
C LIBRARY

C PROGRAMMING SUPPORT PRODUCTS

The list of C support packages is increasing all the time. Whether the software is suitable for your application should be adequately considered before making the investment. Most of the products are designed to be plugged into pre-existing programs. They offer much needed solutions to problems that require extensive hardware and assembly coding. Most products can be combined with the major C compilers.

There are several quality C function libraries on the market. Find out if the functions are supplied in both source and object format, or just in a large library file. Having the source code is a plus, but not every company is willing to sell it. If the source code is provided, it can be extremely tutorial in nature. If you have trouble deciding which package is right for you, write and ask for a list of the functions with a short description. Certain packages are geared toward graphics, while others focus on applications and internal functions. Some have a sprinkling of all three types.

Even though the support software is mentioned here in terms of interfacing with C, most of them can be merged with Assembly, Pascal, or BASIC.



BASIC TO C TRANSLATORS

S-TRAN

Unix users should be interested in S-TRAN. The package accepts popular BASIC dialects and produces functionally equivalent C programs that will execute under Unix. S-TRAN reads in BASIC source code and outputs C source code suitable for your compiler. Maintenance to the original software system (an accounting package, for example) could be made in BASIC and retranslated or altered in the new C version.

S-TRAN ($1500)
Software Manufacturers, Inc.
4009 Pacific Coast Highway
Torrance, CA 90505
(213) 316-2220

BASTOC

JMI Software has developed a BASIC to C converter that covers a wide range of applications. ANSI standard BASIC is support as input, and the C source output is suitable for a wide range of compilers. Formatted and structured C code is produced. The conversion system can be tailored for particular forms of BASIC. The BASTOC translator is written in C.

BASTOC consists of a translator program and a library of run-time functions. Once the BASIC program is converted to C, and compiled, the object module is then linked with a BASTOC run-time library to form the final executable version. BASTOC can run on the IBM PC with PC-DOS, any compatible computer, or a TRS-80 model 16 with Xenix.

BASTOC ($300)
JMI Software Consultants, Inc.
1422 Easton Road
Roslyn, PA 19001
(215) 657-5660

WINDOWING

WINDOWS IN C

Creative Solutions has put together an inexpensive windowing library. The functions take care of the overhead involved with fancy tiling on the screen. Text windows — horizontal or vertical — can be opened and made active. The text in the windows can be paged up and down or sideways. The active window overlaps the inactive ones. Windows in C provides complex screen management for multiple text windows, with assembly-level speed and at a reasonable price. The routines use line-oriented DMA for the screen work. Any number of windows can be created with each one attached to a separate text file, and windows can be moved across the display area. The documentation is clearly written and includes sample programs.

Windows in C
Creative Solutions
21 Elm Ave.
Richford, VT 05476
(802) 848-3710

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

VIRTUAL SCREEN INTERFACE

Amber Systems has developed an extensive set of window managing routines for compiled C, BASIC, FORTRAN, and Pascal programmers. The VSI (Virtual Screen Interface) package has been tested on the IBM PC, Compaq, Corona, Eagle PC, and Hyperion. The interface for C is considered to be a Lattice Level 1 version; lower level assembly access (Level 0) is possible. Up to 255 windows can be managed, as well as 64K of data space. An interactive, debugging tool comes as part of the package, allowing you to test command lines that determine window functions.

VSI Amber Systems
811 Alderbrook Lane
Cupertino, CA 95014
(408) 996-1883

LATTICE WINDOWS

Especially for the Lattice C programmer, Lifeboat has its own "Window" subroutine library for manipulating "virtual" screens. A single program can manage one window, or each separate task within a program can have its own window. The Level 1 Lattice interface is used to handle the window calls. Virtual windows may be opened, closed, written to, read from, and moved about the screen.

Lifeboat Associates
1651 Third Ave.
New York, NY 10028
(212) 860-0300

INTERPRETIVE C

INSTANT-C

Finally, an interpreter for C. By the time this issue reaches you, Rational Systems should have their C interpreter, Instant-C, on the market. Essentially, you write your C programs in an editor and then execute the whole program (or just one function) two seconds later. Symbolic debugging and single stepping are also possible. When the interpreter finds an error, it returns you to the editor and places the cursor on the offending line. Instant-C follows K&R and works with PC-DOS, MS-DOS, or CP/M-86. Pre-existing C programs can be interactively driven, and any Instant-C program can be saved in source format or as an executable file. The tedium of editing, recompiling and relinking is over. If you change just one line of your program, you can see the results instantly.

Instant-C ($500)
Rational Systems, Inc.
P. O. Box 506
Natick, MA 01760
(617) 653-6194

FUNCTION LIBRARIES

HALO

This library is primarily oriented toward graphic functions. The functions come in a .LIB format only. No source code other than a sample program, is included. The HALO library cannot be unbundled to create smaller libraries or to attach particular HALO functions to a larger library. An excellent interpretive program included with the package allows you to experiment interactively with the graphic calls. You can try one HALO command after another and watch it create shapes on a suitable color screen. Lines, circles and text can be colored.

You could, perhaps, view the package as an extended version of BASICA's graphics routines. If you are trying to develop color graphics in C, you will have immediate access to routines that up until now could only be utilized through BASICA. HALO has additional functions such as cross-hair drawing and texture/cross-hatch utilities much like the Apple. Bar graphs, boxes, circles, and even animation (moving areas) can be called. The HALO package is accessible from BASIC (compiled and interpretive), C, FORTRAN,Pascal and Assembler.

HALO ($150)
Lifeboat Associates
1651 Third Ave.
New York, NY 10028
(212) 860-0300

CTOOLS AND CTOOLS2

Doing professional application programming is an exacting and time-consuming endeavor. Blaise Computing produces professional tools to ease this enormous task. (See a review of TOOLS and TOOLS2 elsewhere in this issue.) CTOOLS is especially designed to solve the problems normally encountered in application work: string handling (insert, extract, convert); screen I/O (video mode, cursor control, window-style scrolling, attribute display); graphics (read a point, write a point, draw lines); general utility (block memory moves, hi-low byte access, abort routine); and display/prompt features. CTOOLS2 caters to DOS 2.0 and does the difficult file manipulation work: create, open, close, write, delete, mkdir, rmdir, disk drive selection, search directory, etc. It also contains a substantial array of memory allocation and interrupt vector routines. One routine will take care of running a DOS command (DIR, CHKDSK, etc.) under program control, while another will load and execute a secondary program. The documentation is well written and the demonstration programs illustrate the instantaneous screen work you can easily create with these CTOOLS.

Blaise markets one other C application support package,

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

EXEC. It is a monitor utility that takes care of your program chaining problems. You are free to chain at will among numerous programs. They also sell Microsoft's Pascal compiler, which many prefer to IBM's version.

CTOOLS ($125)
CTOOLS2 ($100)
Blaise Computing, Inc.
2034 Blake St.
Berkeley, CA 94704
(415) 540-5441

C FOOD SMORGASBORD

Just as the name implies, this package is a collection of utilities for the C programmer. The smorgasbord includes numerous math functions, 16 console I/O routines, BIOS access (video, some color).

C FOOD SMORGASBORD
Lifeboat Associates
1651 Third Ave.
New York, NY 10028
(212) 860-0300

GREENLEAF FUNCTIONS

One of the most extensive function libraries available for all major C compilers is from Greenleaf Software in Carrollton, Texas. You receive over 65 C source files as well as the complete assembler code for three .LIB files. You need two double-sided diskettes to store the files. The functions perform the most requested tasks C programmers face: keyboard control, extensive string and printer utilities, graphics routines, cursor functions, Date and Time, RS232 access, DOS 2.0 file handling, SYSINT caller, and several equipment and math functions. Two of the most interesting routines are a STACK dump diagnostic function and a set of scroll-up and scroll-down utilities. A few of the simpler C routines you could easily write yourself, but from purely a financial standpoint, the package would pay for itself the first time you needed one of the more advanced functions. Just as a tutorial it is excellent.

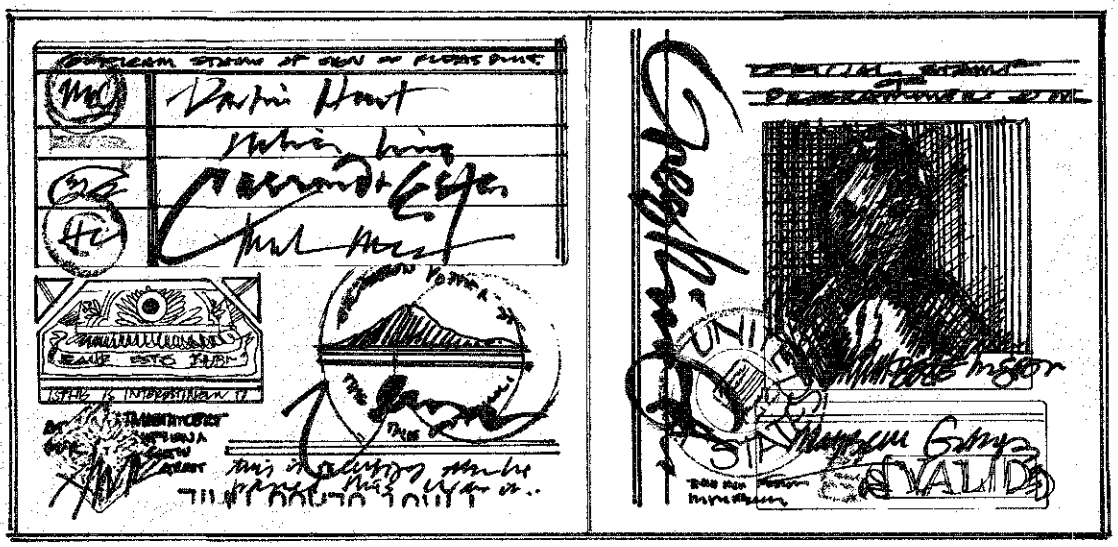
Here is a well put together package from an independent software developer. You receive a clear manual, quick reference card, complete source code, and free subscription to the Greenleaf Newsletter. Several demo programs illustrate how most of the functions are to be used.

Greenleaf Functions ($175)
Greenleaf Software
2101 Hickory Drive
Carrollton, TX 75006
(214) 446-8641

C TOOLS

XOR Corporation has written a series of tools that directly support the CI-86 compiler. Their C routines cover a broad range of needed functions: keyboard handling, random number generation, text-mode screen work, graphics primitives, sound routines, and system software enhancements. Both source and OBJ files are included.

C TOOLS
XOR Corporation
5421 Opportunity Court
Minnetonka, MN 55343
(812) 938-0005



OFFICIAL pj LIBRARY CARD

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

C BUILDING BLOCKS

CSE - C SCREEN EDITOR

Northwest Microsystem Design markets a text editor (CSE) for the IBM PC that is written in C. The best part of the deal is the fact that the complete C source code is included. Besides its value as a text editor, CSE serves as an excellent tutorial for creating your own text processing tools. Macros and function key redefinition are supported by the editor.

C Screen Editor - CSE ($60)
Northwest Microsystem Design
P. O. Box 10853
Eugene, OR 97401
(503) 484-7129

BASIC_C

There is a great deal of interest in the BASIC_C package from C Source in Kansas City. The first question usually asked is "What exactly is it?" BASIC_C could be described as the BASIC programmer's touch-stone to the C language. Rather than fighting the difficult C syntax, with BASIC_C you can write 90 percent of your code in a BASIC style format. You can code merrily along as if you were in BASIC, remembering to put a ";" at the end of your statements. Your variables do need to be declared as "int" or "char" or whatever.

With appropriately used "#define" statements, you can substantially alter the look of C code, and, among other things, that is exactly what C Source has done to produce their package. You get the feel for calling one function after another which is a standard practice in C. It would be interesting to see if someone doesn't write a "pre-processor" for BASIC programs that creates source code acceptable to BASIC_C. The result would be a BASIC to C translator with the hardest part done by C Source. Judging from the examples provided with BASIC_C, it would not be a formidable task. If you want the easiest transition to C, take a closer look at BASIC_C.

BASIC_C ($175-$375)
C Source
12801 Frost Road
Kansas City, MO 64138
(800) 821-7700, Ext. 303

PANEL

Custom input screens suitable for data entry are created with PANEL. After the input screen is drawn and the data types are defined, PANEL generates the appropriate source code for your compiler: Lattice C, PL/I-80, PL/I-86, COBOL, or Pascal.

PANEL ($350)
Lifeboat Associates
1651 Third Ave. New York, NY
10028 (212) 860-0300

C HELPER

A combination of C development utilities would best describe this package. Flowchart programs, a C code beautifier, XREF, DIFF and other support utilities are provided with the source code. Both CP/M and MS-DOS versions are available.

C HELPER ($135)
Computer Innovations, Inc.
980 Shrewsbury Ave.
Tinton Falls, NJ 07724
(201) 542-5920

FLOAT87

The floating-point 8087 chip is supported using this function library. Both Intel .OBJ and Microsoft .LIB formats can be requested. The routines interface with Lattice/MS C and PL/I-86.

FLOAT87 ($125)
Lifeboat Associates
1651 Third Ave. New York, NY 10028
(212) 860-0300

dBASE II AND C

Computer Innovations has a unique tool that more companies should create: functions to access major database manager files. Their current tool is a C_to_dBase II package that allows you to directly manipulate dBase II records. If you have lamented the speed of dBase II, a high-level answer is here. The C_to_dBase II interface is a complete set of file handles, both sequential and random. The source code is included for alteration or as a tutorial. It could be used as a starting point for a custom file management tool. The code is set up for their CI-86 compiler, but it is written in a generic format that should allow easy conversion to the other standard C compilers.

C_to_dBase II ($95)
Computer Innovations, Inc.
980 Shrewsbury Ave.
Tinton Falls, NJ 07724
(201) 542-5920

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PHACT

An index file management library, PHACT has a balanced BTree structure as well as multi-field keys, sequential or random processing, record lockout, and variable length records. The library is designed to interface with CI-86, under MS-DOS or CP/M-86.

PHACT ($250)
Computer Innovations, Inc.
980 Shrewsbury Ave.
Tinton Falls, NJ 07724
(201) 542-5920

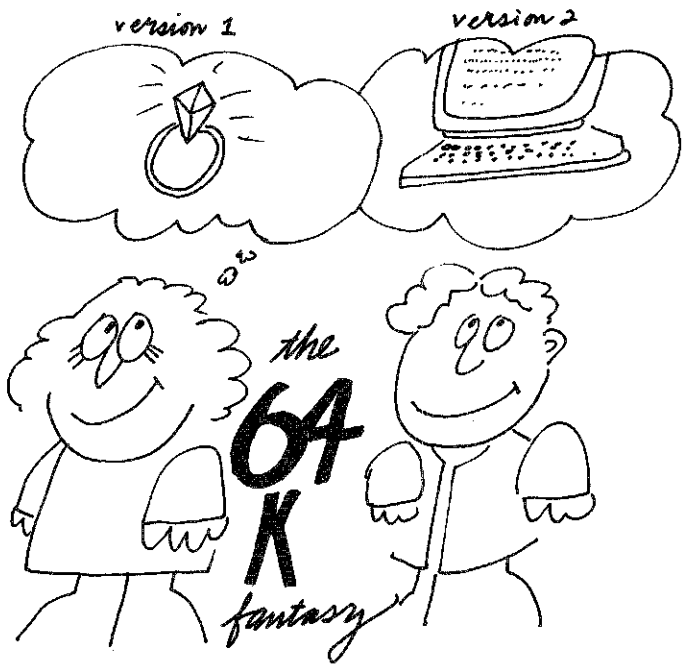
DATABURST

The chore of data entry and input screen maintenance is taken care of with Key Solutions package, DataBurst. Use any text editor to create an input screen and Databurst translates it into a format file which it later uses for data entry and validity checking. A compiled and interpretive BASIC version is available, and a C source code generating version is now ready. All display attributes are supported, as well as complete error checking. DataBurst brings mainframe style screen control to micros.

DataBurst ($225)
Key Solutions
P. O. Box 2297
Santa Clara, CA 95055
(408) 554-6125

BTRIEVE

SoftCraft Inc. combines every possible program interface with their record management utility, BTRIEVE. The package has interfaces for C, Pascal, COBOL, BASIC (Interpretive and compiled), and Assembly. You need not pay extra for the additional languages. BTRIEVE might be called a programmers database manager, since you are in the drivers seat. It handles multi-key access, index file maintenance, file integrity, relationships among files, I/O cache buffering, and unrestricted file sizes. Program utilities are there which let you load or unload an ASCII file into a BTRIEVE file. Complete record manipulation is straight forward: open file, close file, insert record, update record, delete, get equal, get next, get previous, get greater than, get highest, get lowest, show stats, partition a file over two diskettes, get directory, and abort transaction. The manual is well written and easy to follow in case you are worried about sinking waist deep into ISAM-style cryptic phrases. Besides the first-rate documentation and numerous examples, the package concentrates on providing a safe environment when dealing with keyed records. BTRIEVE is an extensive file and record handling package.



BTRIEVE ($145)
SoftCraft Inc.
P.O Box 9802 #590
Austin, TX 78766
(512)346-8380

BBS

If you have a modem, you should regularly cruise the following BBS's just for the purpose of downloading the latest in interesting C source code. On several of the bulletin boards you can find an excellent XREF, a GREP utility, file dump programs, and the like. The first three Boards listed run what we consider to be the finest on-line, user-friendly BBS software available. It is fast, self-explanatory, and nicely featured. You can log on and request a list of files that have been uploaded since your last visit.

Gene Plantz (312) 882-4227
Gene Plantz (312) 882-4228
Tulsa, OK(918) 749-0718
Virginia (703) 321-7003
Concord, CA (415) 689-2090
Hackernet(617) 791-1957
Computer Innovations (201) 542-5059

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

MISC.

DEBUG AND DEBUG87

A new debugging utility has been developed for 8087 software developers by Dan Rollins, noted assembly language programmer and book author. You need PC-DOS 2.0 and the supplied DEBUG program. The DEBUG87 program is merged with DOS utility, creating an enhanced version that is valuable even if you do not have an 8087 chip. The features include: clear screen function, hex to binary/decimal conversions, 18-digit decimal display of the STACK registers, direct input of decimal numbers into the 8087, special trace option, and execute one subroutine or opcode.

DEBUG87 ($50)
RollinSoft
P.O. Box 12147
La Cresenta, CA 91214
(818) 246-5021

NEW MICROSOFT ASSEMBLER

Microsoft has released a new version (1.25) of their Macro Assembler. At some point IBM may update to their new release, but it is not known when that will take place. The new Microsoft version requires DOS 2.0 and includes a number of enhancements. Besides handling Path names and supporting the 8087 chip, it has a copy of DEBUG, LINKER, XREF, and their LIB maintenance program. Many programmers have been trying to beg, borrow, or steal the MS Library Utility program that allows you to add and delete functions; it now comes as part of their Macro Assembler. The price is still $100. Pick one up at your local Microsoft software dealer.

UNIX ON THE PC

The Microsoft Xenix version of Unix should now be up and running on the PC. The Santa Cruz Operation in California did the porting. For up-to-date information about the implementation and system requirements, contact them directly.

The Santa Cruz Operation
500 Chestnut St.
Santa Cruz, CA 95060
(408) 425-7222

HARD COPY TO DISKETTE

Spending a lot of time transcribing hard copy into your word processor? Information Services will do the job for you for just $.50 a page. They use an optical reader to do the translation. Find out which type fonts it reads, and you may save yourself hours of typing.

Information Services, Inc.
Virginia Beach, VA
(804) 427-5200

MAGAZINE INDEX

For $40 you can have a complete database of IBM PC related magazines. The table of contents, with a brief description of the articles, is included for BYTE, PC, PC TECH, PC WORLD, PCA, and SOFTALK (what no, PJ?). The $40 includes three diskettes and the associated files. The database is in a PC-FILE format, so you just use Jim Button's file manager to access them. Letus A-B-C (R. Nelson) is providing the user-supported service. You can also send Letus three DS-DD diskettes with a return mailer and $10 per diskette to reduce the price. Copying, as with any Freeware, is encouraged.

Letus A-B-C ($40)
3790 El Camino Real,
Suite 2006
Palo Alto, CA 94306

THE INSIDE STORY

If you have developed a software package, or are in the process of creating a package for the IBM PC, you should contact Microsoft and become part of their ISV (Independent Software Vendor) program. You need only write and tell them you are an independent software writer; they will then put you on their ISV list and you will receive inside information about DOS, EXEC functions, MS-Tips, Software updates, Windows, and their newsletter.

Ken Sanders/ISV program
Microsoft Corp.
10700 Northup Way
Bellevue, WA 98004
(206) 828-8080

MONOCHROME GRAPHICS

So you thought that the IBM Monochrome Display had no graphic capabilities, or that it was a set-in-concrete monitor. Just type in the following program and see if it won't change your mind just a little. You could always run it on a friend's system when he is not looking and watch him sweat blood, thinking his display has gone on the fritz.

```
100 cls
110 key off
120 width 80
200 out &H3B4,8 : OUT &H3B5,3
210 out &H3B4,1 : OUT &H3B5,35
220 out &H3B4,2 : OUT &H3B5,63
240 PRINT ' HOW DOES THIS LOOK '
260 DEF SEG =&HB000
270 FOR I = 3 TO 38
280 FOR J = 0 TO 29
290POKE 80*I+2*J,68
300 NEXT J
310 NEXT I
330 LIST
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

ARE YOU AN ISSUE SHY OF A FULL SET?

Complete your Programmer's Journal library! Order any back issues you may have missed — or become a new subscriber. Use the form below or the handy subscription envelope provided. You'll be glad you did.

- Vol. 1, No. 1: Norton Pascal I/O, Program Style Manual, COBOL, System Analysis, BASIC modules, Rounding error.
- Vol. 1, No. 2: Defining character sets, C86 BIOS calls, Copy protection, Pascal code, Input routines, Writing an Assembler.
- Vol. 1, No. 3: ASM dumb terminal, C source code, ASM graphics line drawing, Text editors reviewed, Utilities.
- Vol. 1, No. 4: Diskette directory via Pascal, MASM WHILE-WEND, C/ASM interface, ASM stay resident routine, Structured programming.
- Vol. 1, No. 5: Multi-tasking software review, Transporting C Programs, Programmers calculator, dBase bargraphs, Utilities.
- Vol. 1, No. 6: ASM printer configuration, DOS patch utility, Hi-speed graphics, Software ownership, dWINDOW review.
- Vol. 2, No. 1: Freelance consulting, Sorts in C, Pascal text editor source code, Designing national database, Ultra utilities.

Name _____

Company _____

Address _____

City _____ State _____ Zip _____

Telephone _____

Send me the following back issues at $4.00 each:

_____ Vol. 1, No. 1
_____ Vol. 1, No. 2
_____ Vol. 1, No. 3
_____ Vol. 1, No. 4
_____ Vol. 1, No. 5
_____ Vol. 1, No. 6
_____ Vol. 2, No. 1
_____ Total

_____ 1 year bi-monthly subscription
at $24.00 ($30 Canada, $40 other)

_____ Check enclosed _____ Bill me

_____ Charge my Bankcard: ___ VISA ___ MC ___ Am Ex

_____ Expir. Date _____

Signature _____

Send to:
PROGRAMMER'S JOURNAL
P.O. Box 30160
Eugene, OR 97403
(503) 345-3043

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

DON'T MISS C

"moves at a good pace."

"C provides pointers."

"straightforward..."

"no garbage..."

"an evolving language."

"powerful..."

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED